Storing XML on Relational Tables

Federico Ulliana
GraphIK, LIRMM, INRIA

Slides collected from James Cheney and Sam Idicula

Boston, winter '99 : XML standardization

Jan 2000 : people wondering …

Now, how can I publish online my relational data?
(XMLAGG - Xperanto; Mappings - SilkRoute)

Boston, winter ’99 : XML standardization

Jan 2000 : people wondering …

Now, how can I publish online my relational data?
(XMLAGG - Xperanto; Mappings - SilkRoute)

Feb 2000 : people (again) wondering …

I created my first10GB XML document crawling web data.
Now, how can I query it ?

3 schools for processing XML data

1. Flat streams: store XML data as is in text files
    - query support: limited; fast for retrieving whole documents

2. Native XML Databases: designed specifically for XML
    - XML document stored in XML specific way
    - Goal: Efficient support for XML queries

3. Re-use existing DB storage systems
    - Leverage mature systems (DBMS)
    - How ? Map XML document into flat tables

Why transform XML data into relations?

Native XML databases need:

- storing XML data, indexing,
- query processing/optimization
- concurrency control
- updates
- access control, . . .
- Nontrivial: the study of these issues is still in its infancy – incomplete support for general data management tasks

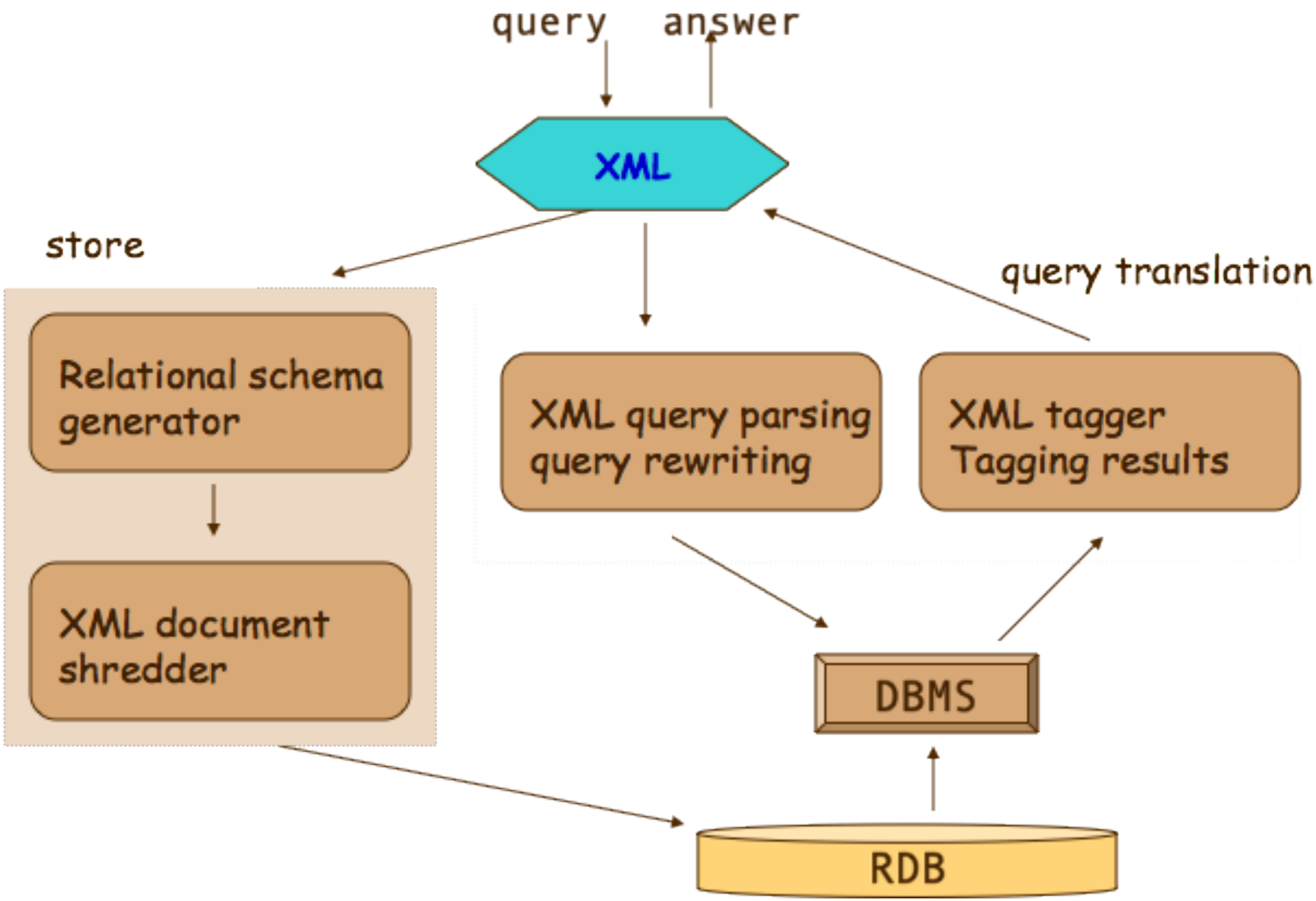
Haven't these already been developed for relational DBMS!?

- Why not take advantage of available DBMS techniques?

From XML to relations : steps

1. Derive a relational schema
2. Insert XML data into relational tuples
3. Translate XML queries to SQL queries
4. Convert query results back to XML

Architecture



Nontrivial issues

Data model mismatch

- DTD: recursive, regular expressions/nested content
- relational schema: tables, single-valued attributes

Information preservation

- lossless: there should be an effective method to reconstruct the original XML document from its relational storage
- propagation/preservation of integrity constraints

Query language mismatch

- XQuery, XSLT: Turing-complete
- XPath: transitive edges (descendant, ancestor)
- SQL: first-order, limited / no recursion

Plan

Schema-unaware

Schema-aware

Commercial solutions

SCHEMA-UNAWARE XML STORAGE

Schema-unaware storage

Storage easier if we have a fixed schema

But, often don't have schema

Or schema may change over time

- schema updates require reorganizing or reloading!

So: schema-oblivious XML storage

Schema chaos: In this scenario, customers want the flexibility to manage XML data that may or may not have schema, or may have "any" schema. For instance, a telecommunication customer wants to manage XML data generated from different towers, which generate documents with slightly different schemas from each other. They want to store them in one table and perform efficient query on the shared common pieces.

The basics first

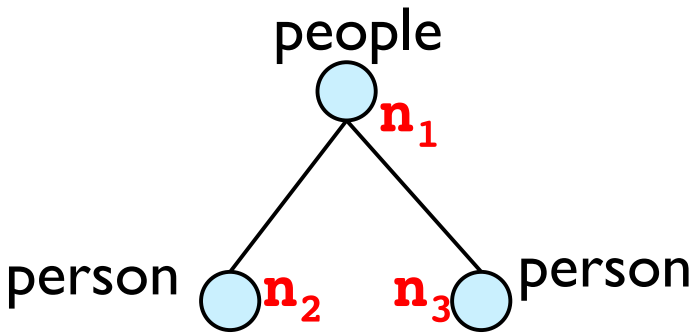
"before thinking about sophisticated solutions, how the simplest and most obvious approaches perform?"

Round 1) EDGE vs VERTICAL-EDGE

EDGE storage

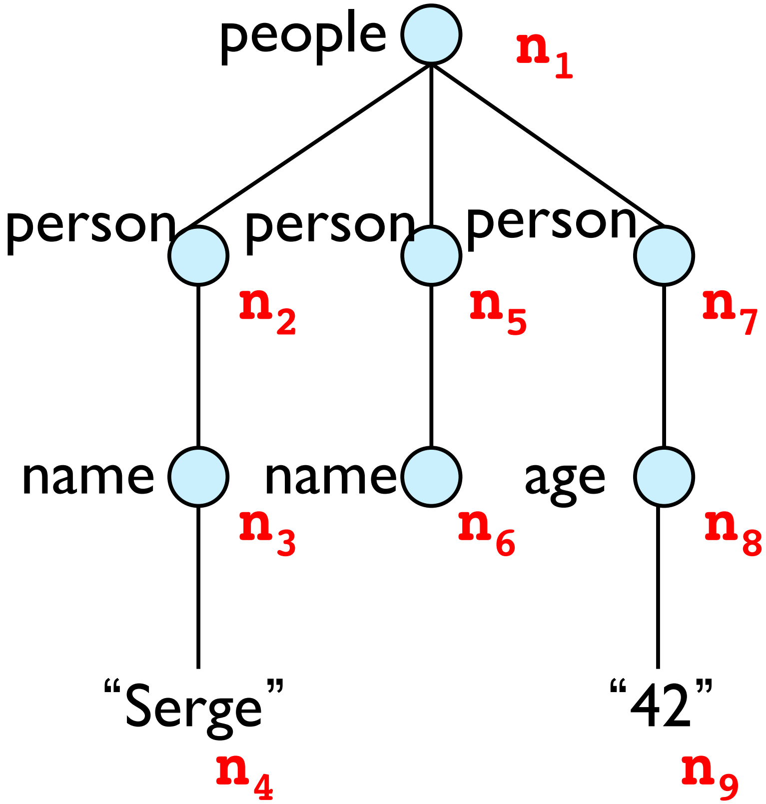
Observation: XML <u>ordered</u> trees can be encoded with

binary relation	`EDGE(parent,child)`
order relation	`NEXT-SIBLING(prec,succ)`



$\mathtt{EDGE}(n_1, n_2)$
$\mathtt{EDGE}(n_1, n_3)$
$\mathtt{NEXT\text{-}SIBLING}(n_2, n_3)$

Edges & Values



EDGES

source	target	ordinal	tag	type
	n_1		people	elt
n_1	n_2	1	person	elt
n_1	n_5	2	person	elt
n_1	n_7	3	person	elt
n_2	n_3	1	name	elt
n_3	n_4	1		txt
n_5	n_6	1	name	elt
n_7	n_8	1	age	elt
n_8	n_9	1		num

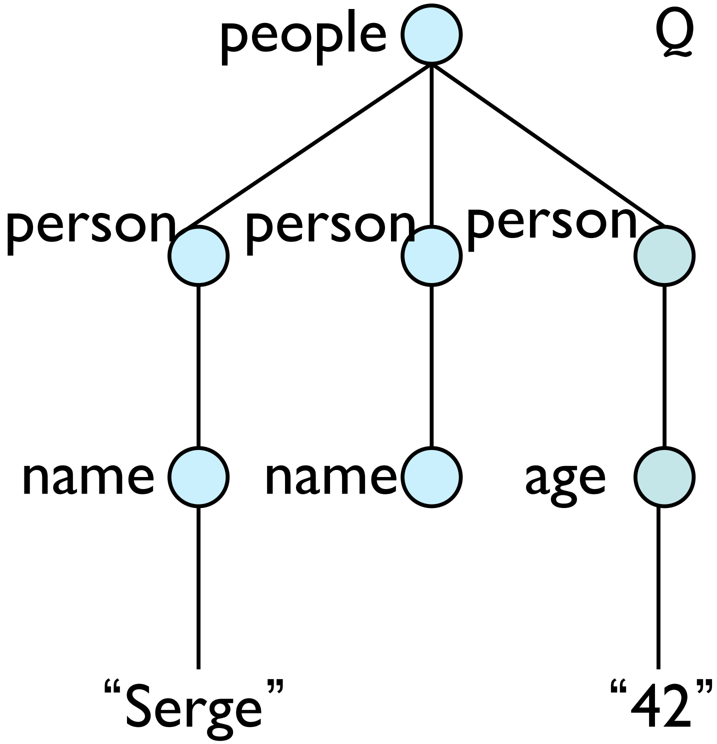
TEXTVALUES

node	value
n_4	Serge

NUMVALUES

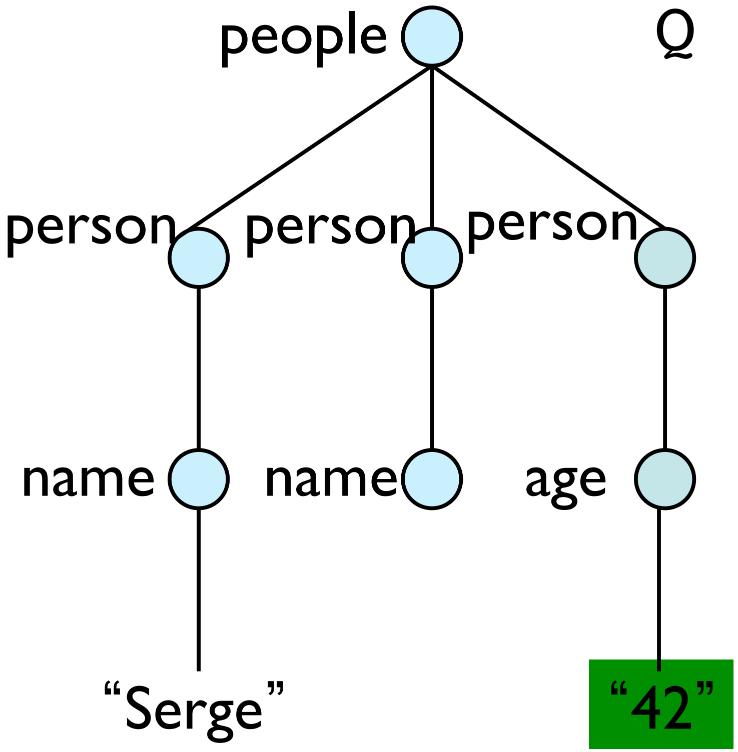
node	value
n_9	42

Querying



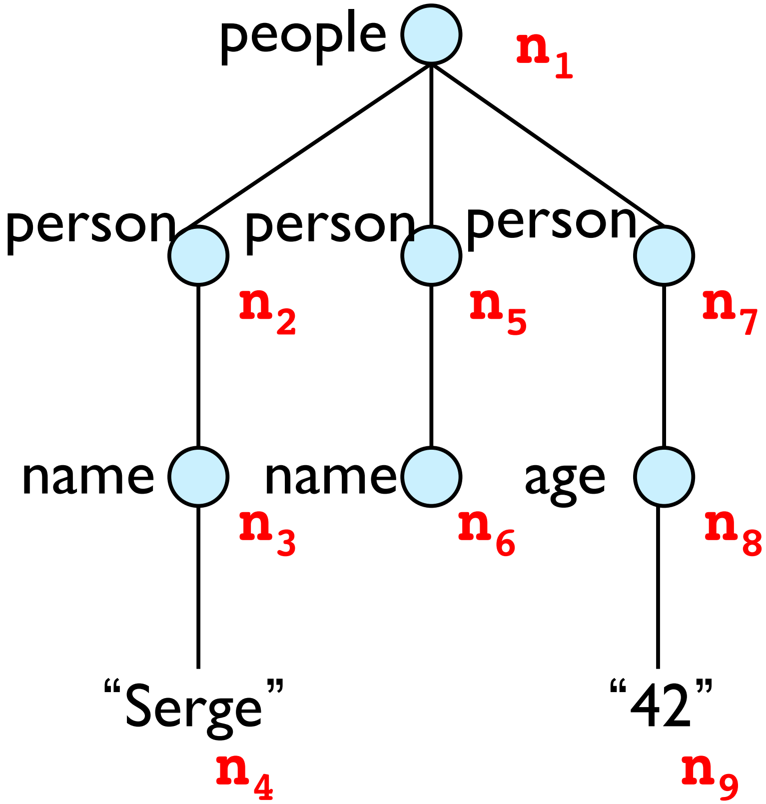

```
Q = /people/person/age/text()
```

Querying



`Q = /people/person/age/text()`

/people/person/age/text()



EDGES

source	target	ordinal	tag	type
	n_1		people	elt
n_1	n_2	1	person	elt
n_1	n_5	2	person	elt
n_1	n_7	3	person	elt
n_2	n_3	1	name	elt
n_3	n_4	1		txt
n_5	n_6	1	name	elt
n_7	n_8	1	age	elt
n_8	n_9	1		num

TEXTVALUES

node	value
n_4	Serge

NUMVALUES

node	value
n_9	42

/people/person/age/text()

```
SELECT N.value
FROM      EDGES as e1
          EDGES as e2
          EDGES as e3
          EDGES as e4
          NUMVALUES N
WHERE
          e1.target=e2.source
AND       e2.target=e3.source
AND       e3.target=e4.source
AND       e1.tag="people"
AND       e2.tag="person"
AND       e3.tag="age"
AND       e3.target=e4.source
AND       e4.type="num"
AND       e4.target= N.node
```

EDGES

source	target	ordinal	tag	type
	n_1		people	elt
n_1	n_2	1	person	elt
n_1	n_5	2	person	elt
n_1	n_7	3	person	elt
n_2	n_3	1	name	elt
n_3	n_4	1		txt
n_5	n_6	1	name	elt
n_7	n_8	1	age	elt
n_8	n_9	1		num

TEXTVALUES

node	value
n_4	Serge

NUMVALUES

node	value
n_9	42

/people/person/age/text()

```
SELECT N.value
FROM     EDGES as e1
         EDGES as e2
         EDGES as e3
         EDGES as e4
         NUMVALUES N
WHERE
         e1.target=e2.source
AND      e2.target=e3.source
AND      e3.target=e4.source
AND      e1.tag=“people“
AND      e2.tag=“person”
AND      e3.tag=“age”
AND      e3.target=e4.source
AND      e4.type=“num”
AND      e4.target= N.node
```

Lots of joins

TEXTVALUES

node	value
n_4	Serge

NUMVALUES

node	value
n_9	42

/people/person/age/text()

```
SELECT N.value
FROM     EDGES as e1
         EDGES as e2
         EDGES as e3
         EDGES as e4
         NUMVALUES N
WHERE
         e1.target=e2.source
AND      e2.target=e3.source
AND      e3.target=e4.source
AND      e1.tag=“people“
AND      e2.tag=“person”
AND      e3.tag=“age”
AND      e3.target=e4.source
AND      e4.type=“num”
AND      e4.target= N.node
```

We also need a query testing for text values (UNION)

TEXTVALUES

node	value
n_4	Serge

NUMVALUES

node	value
n_9	42

Querying

Fragmentation: tree spread across the table

EDGES

source	target	ordinal	tag	type
	n_1		people	
n_1	n_2	1	person	ref
n_1	n_5	2	person	ref
n_1	n_7	3	person	ref
n_2	n_3	1	name	ref
n_3	n_4	1		txt
n_5	n_6	1	name	ref
n_7	n_8	1	age	ref
n_8	n_9	1		num

TEXTVALUES

node	value
n_4	Serge

NUMVALUES

node	value
n_9	42

Querying

Fragmentation: tree spread across the table

Indexes **unaware** of tree structure

EDGES

source	target	ordinal	tag	type
	n_1		people	
n_1	n_2	1	person	ref
n_1	n_5	2	person	ref
n_1	n_7	3	person	ref
n_2	n_3	1	name	ref
n_3	n_4	1		txt
n_5	n_6	1	name	ref
n_7	n_8	1	age	ref
n_8	n_9	1		num

TEXTVALUES

node	value
n_4	Serge

NUMVALUES

node	value
n_9	42

How to improve ?

1. **Vertical partitioning**
   group edges targeting same tag-label

2. **Inlining**
   put text and numeric values in the main table

EDGES

source	target	ordinal	tag	type
	n_1		people	
n_1	n_2	1	person	ref
n_1	n_5	2	person	ref
n_1	n_7	3	person	ref
n_2	n_3	1	name	ref
n_3	n_4	1		txt
n_5	n_6	1	name	ref
n_7	n_8	1	age	ref
n_8	n_9	1		num

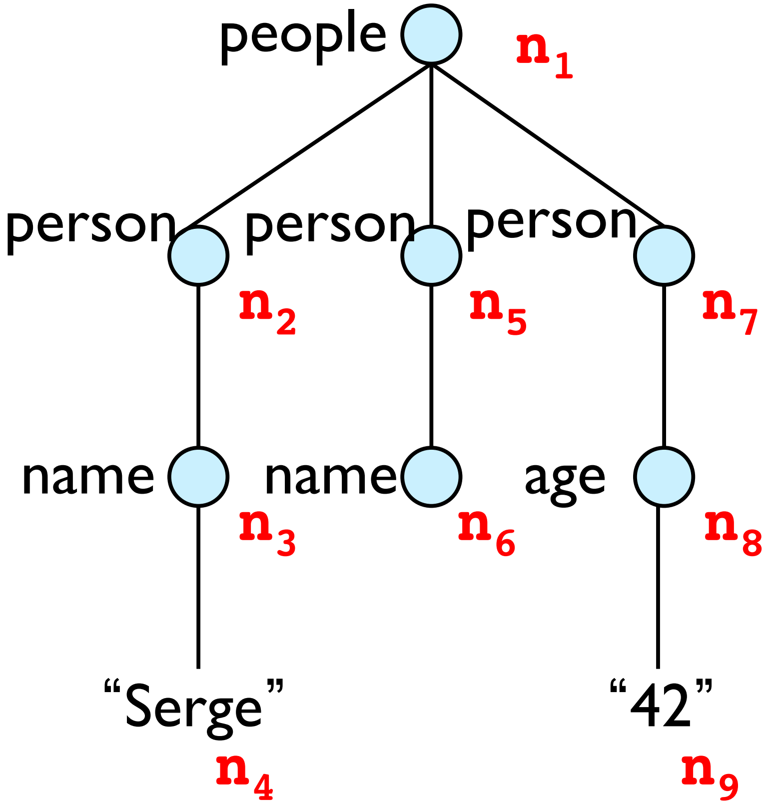
TEXTVALUES

node	value
n_4	Serge

NUMVALUES

node	value
n_9	42

VERTICAL-EDGE + Inline



people

source	target	ordinal	txtval	numval
	n_1			

person

source	target	ordinal	txtval	numval
n_1	n_2	1		
n_1	n_5	2		
n_1	n_7	3		

name

source	target	ordinal	txtval	numval
n_2	n_3	1	Serge	
n_5	n_6	1		

age

source	target	ordinal	txtval	numval
n_7	n_8	1		42

VERTICAL-EDGE + Inline

```
Q = /people/person/age/text()

SELECT   AGE.value
FROM      PEOPLE   P1
          PERSON   P2
          AGE

WHERE
          P1.target=P2.source
AND       P2.target=AGE.source
```

Joins on smaller tables

people

source	target	ordinal	txtval	numval
	n_1			

person

source	target	ordinal	txtval	numval
n_1	n_2	1		
n_1	n_5	2		
n_1	n_7	3		

name

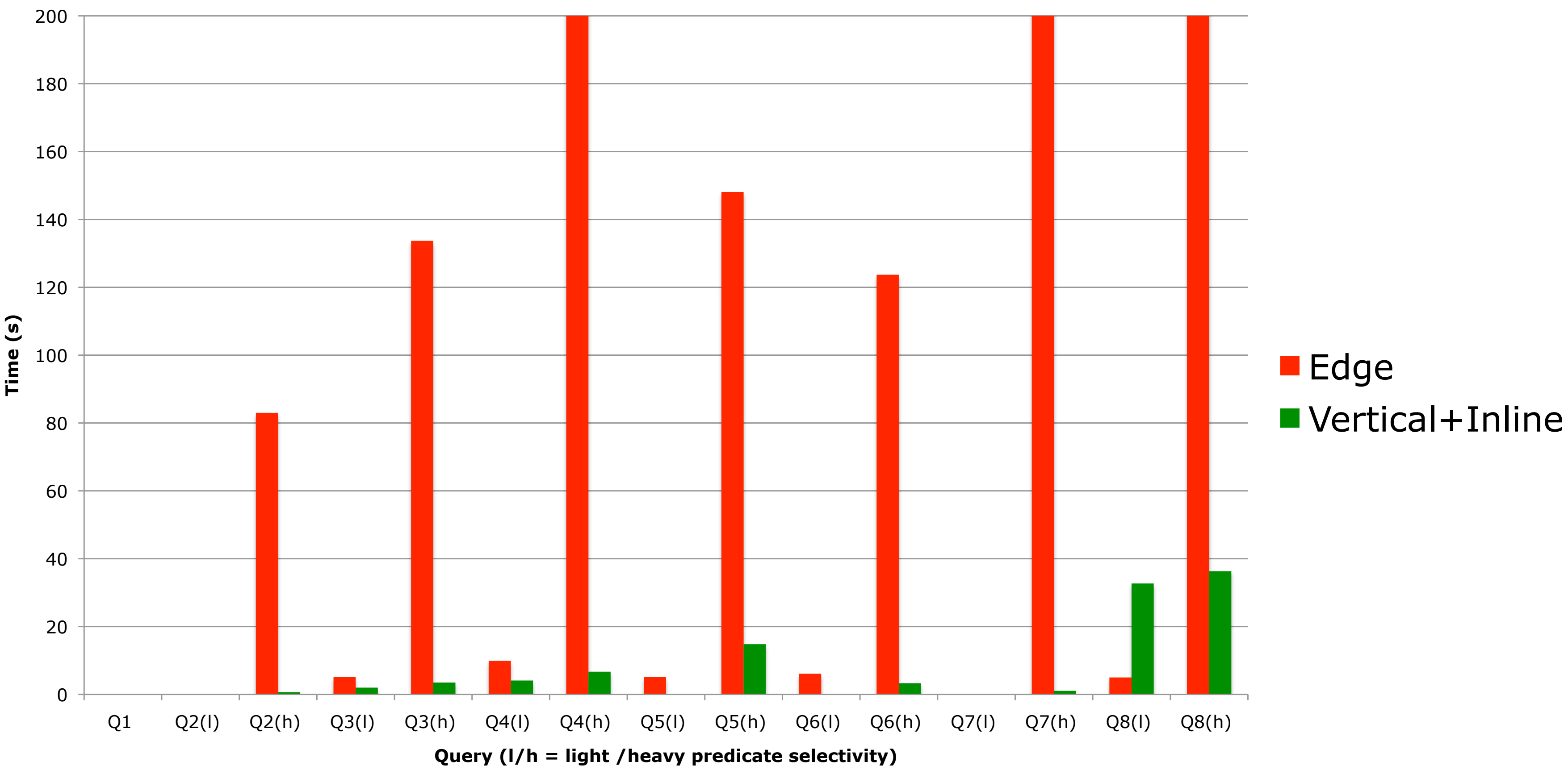
source	target	ordinal	txtval	numval
n_2	n_3	1	Serge	
n_5	n_6	1		

age

source	target	ordinal	txtval	numval
n_7	n_8	1		42

VERTICAL-EDGE+Inline beats EDGE

(query-answering time with the two storages)



The queries `SeFrWh` you cannot ask

- Does it exists a direct flight between Paris and Los Angeles ? ✓
- Does it exists a (possibly indirect) flight between Montpellier and Austin ? ✗
  - problem : we do not know the number of intermediary airports (=joins)
- Does it exists a child for the node N ? ✓
- Is the node M a descendant of node N ? ✗
  - problem : we do not know the depth of a descendant node
  - taking max document depth is not an elegant solution

Issues with XPath axes

`Q = /people//age/text()`

Descendant = implicit recursion
sort of (child)*

Does not translate to
SELECT-FROM-WHERE query

Recursion :
ORACLE, POSTGRES OK
MySQL NO

people

source	target	ordinal	txtval	numval
	n_1			

person

source	target	ordinal	txtval	numval
n_1	n_2	1		
n_1	n_5	2		
n_1	n_7	3		

name

source	target	ordinal	txtval	numval
n_2	n_3	1	Serge	
n_5	n_6	1		

age

source	target	ordinal	txtval	numval
n_7	n_8	1		42

Limits of Edge/Vertical

Indexing unaware of tree structure

- fragmentation : subtree spread across db

Incomplete query translation

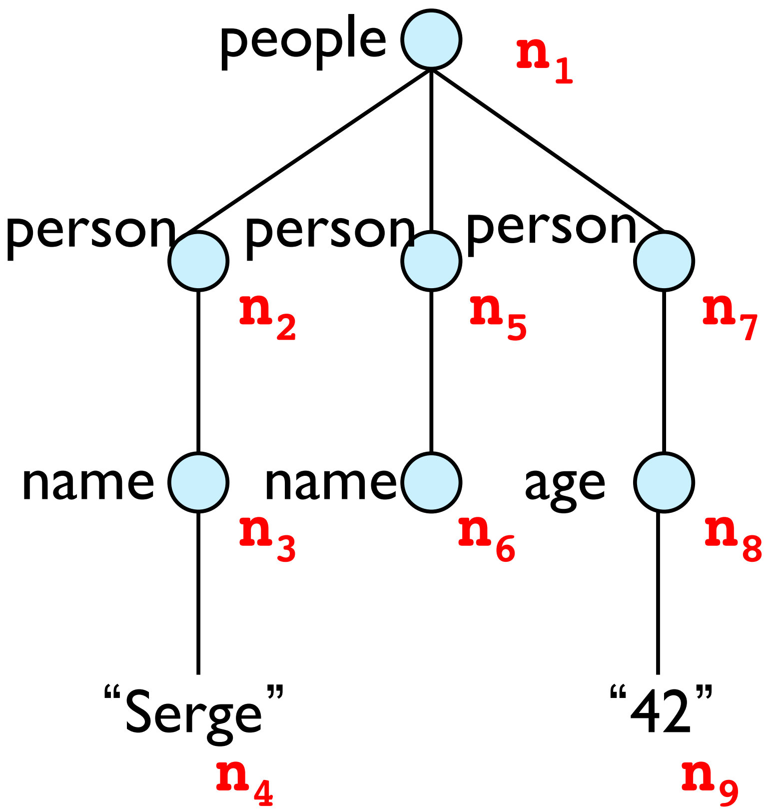
- descendant axis steps involve recursion

Lots of joins

- joins + no indexing = trouble

MONET storage

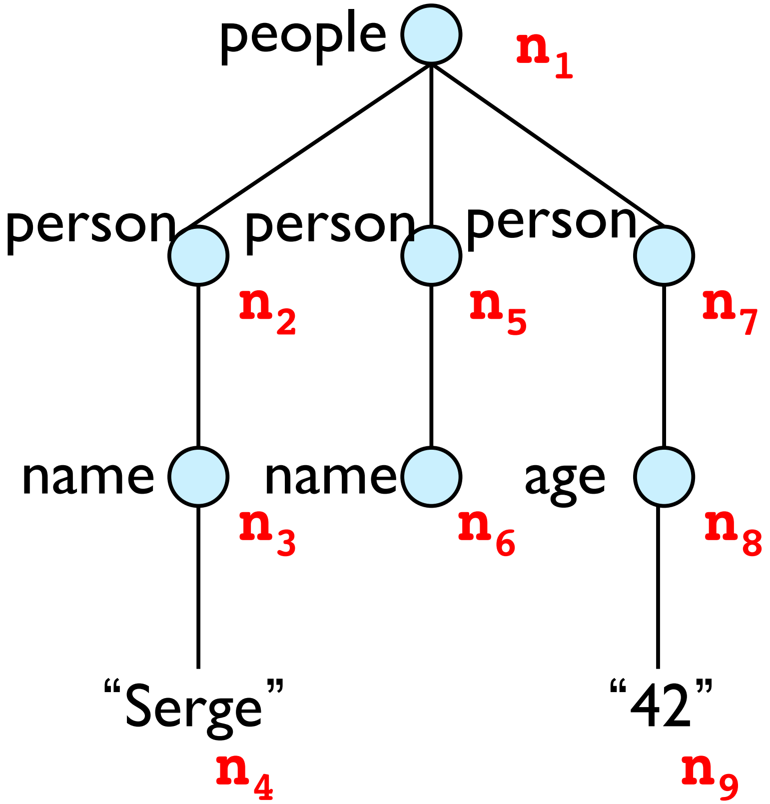
(so called because developed first on Monet-DB)



Idea : one table for each path in the XML tree

- `people`
- `people_person`
- `people_person_name`
- `people_person_age`

MONET storage



people

node	txtval	numval
n_1		

people_person

node	txtval	numval
n_2		
n_5		
n_7		

people_person_name

node	txtval	numval
n_3	Serge	
n_6		

people_person_age

node	txtval	numval
n_8		42

/people/person/age/text()

```
SELECT txtval,numval
FROM
people_person_age
```

people

node	txtval	numval
n_1		

people_person

node	txtval	numval
n_2		
n_5		
n_7		

people_person_name

node	txtval	numval
n_3	Serge	
n_6		

people_person_age

node	txtval	numval
n_8		42

Performances

- MONET (obviously) beats VERTICAL-EDGE+Inlining

Still one question…

And descendant axis ?

```
Q = /people//age
```

How to select the relations to query ?

/people//age

`people_`*(any-seq)*`_age`

`people_`*(any-seq)*`_age`

`people`	`people_person`
✗	✗
`people_person_name`	`people_person_age`
✗	✓

```
//person//*
```

(any-seq)`_person_`*(any-seq)*`_`*(any-tag)*

(any-seq)`_person_`*(any-seq)*`_`*(any-tag)*

`people`	`people_person`
✗	✗
`people_person_name`	`people_person_age`
✓	✓

(any-seq)`_person_`*(any-seq)*`_`*(any-tag)*

```
SELECT node
FROM people_person_name
                                       UNION
SELECT node
FROM people_person_age
```

`people_person_name` `people_person_age`

✓ ✓

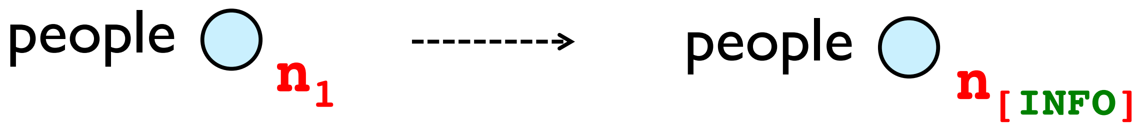
And the remaining axes ?

Maybe we need some new ideas…

INTERVALS

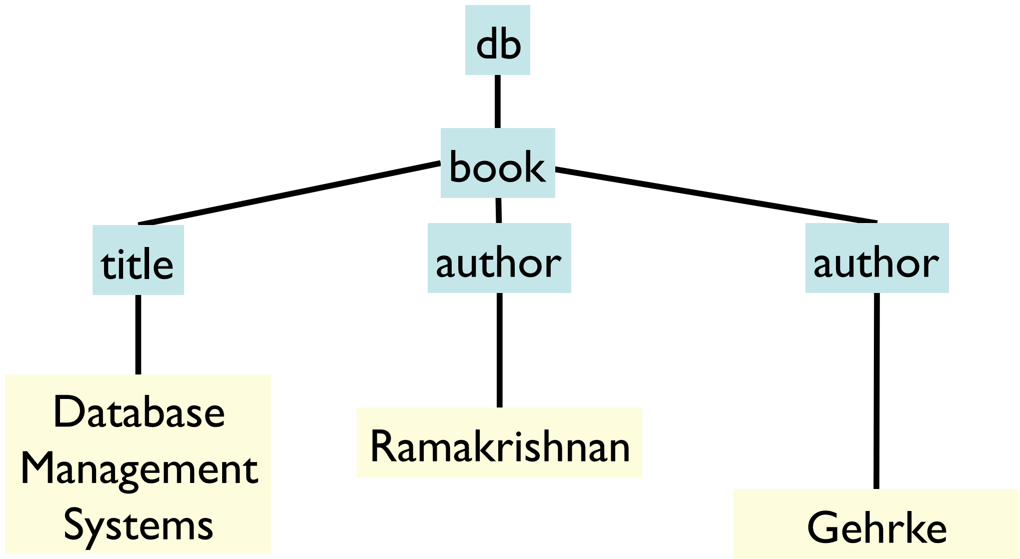
Intervals

Idea: Node-identifier embed navigational-information



Intervals

Think of XML as a linear string

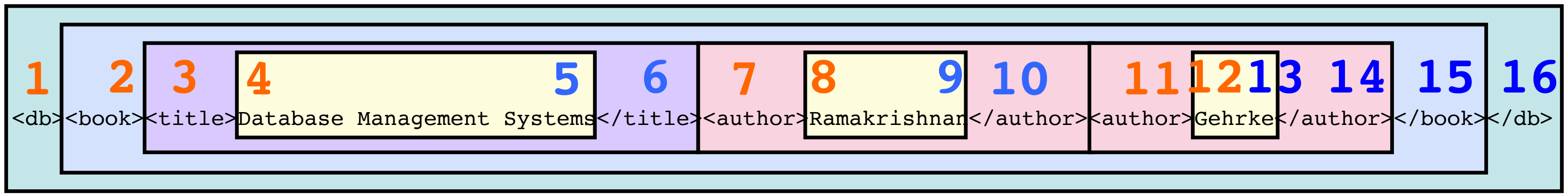


`<db><book><title>Database Management Systems</title><author>Ramakrishnan</author><author>Gehrke</author></book></db>`

Intervals

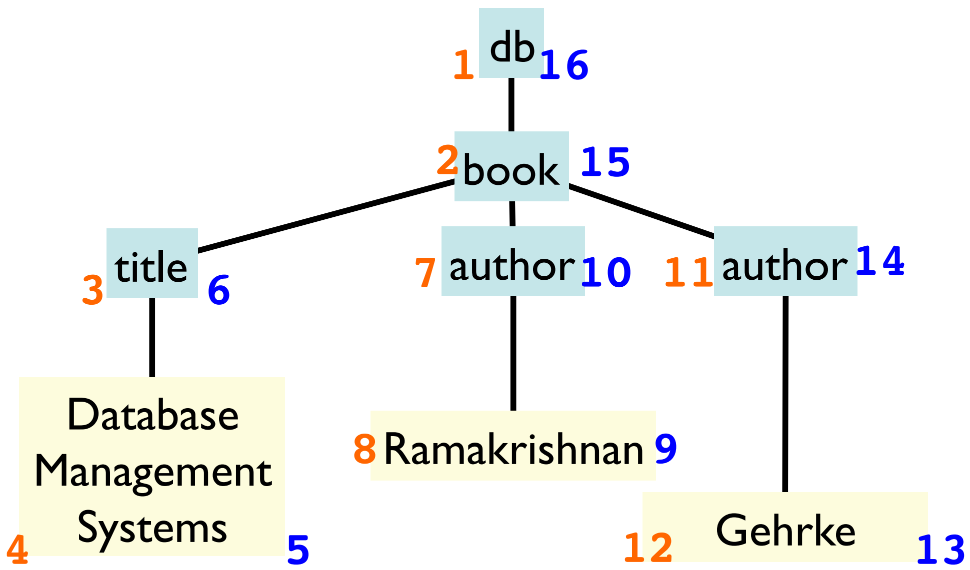
Begin : the first time we see a node (opening tag)

End : the last time we see a node (closing tag)



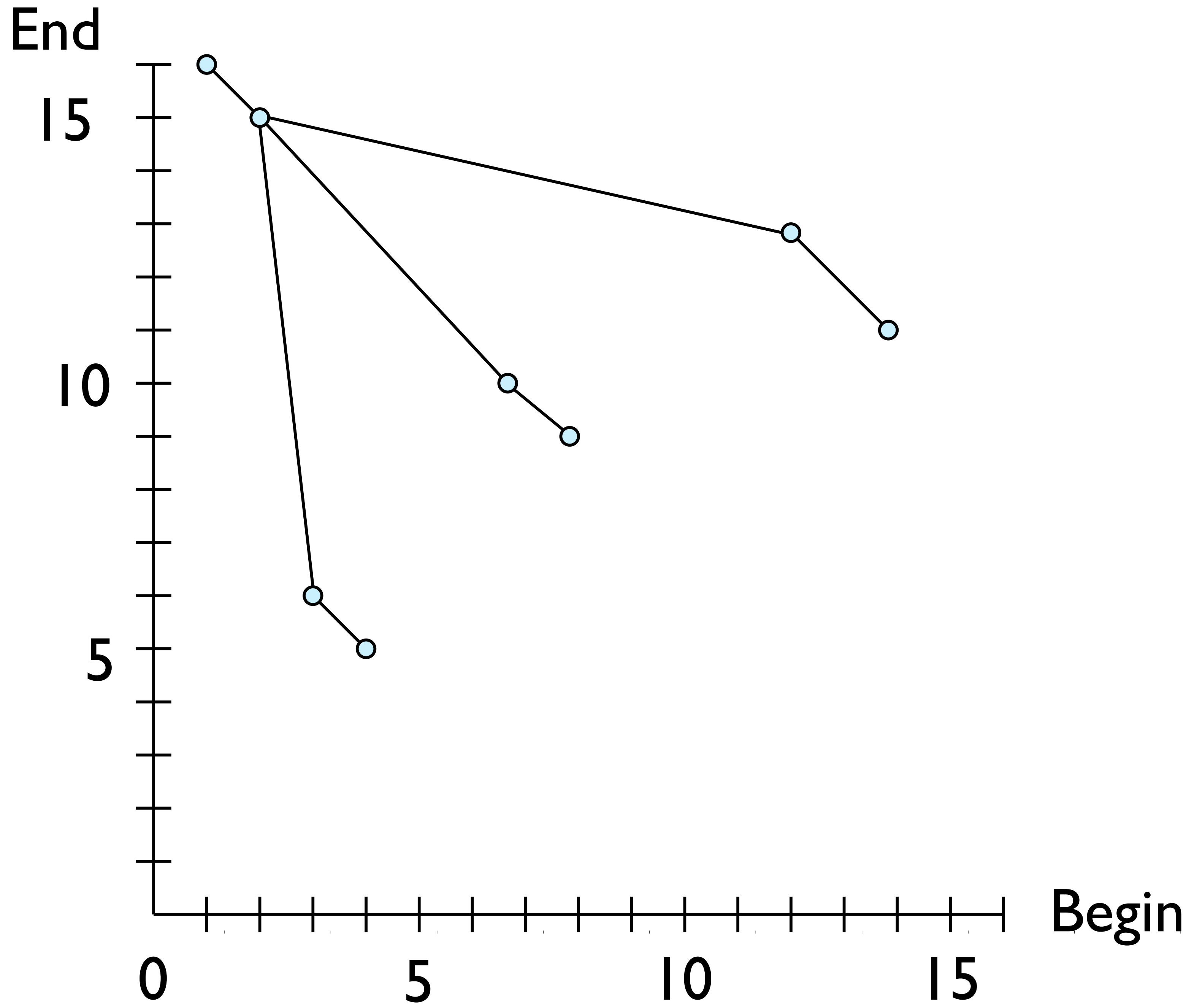
Each node corresponds to an interval on line

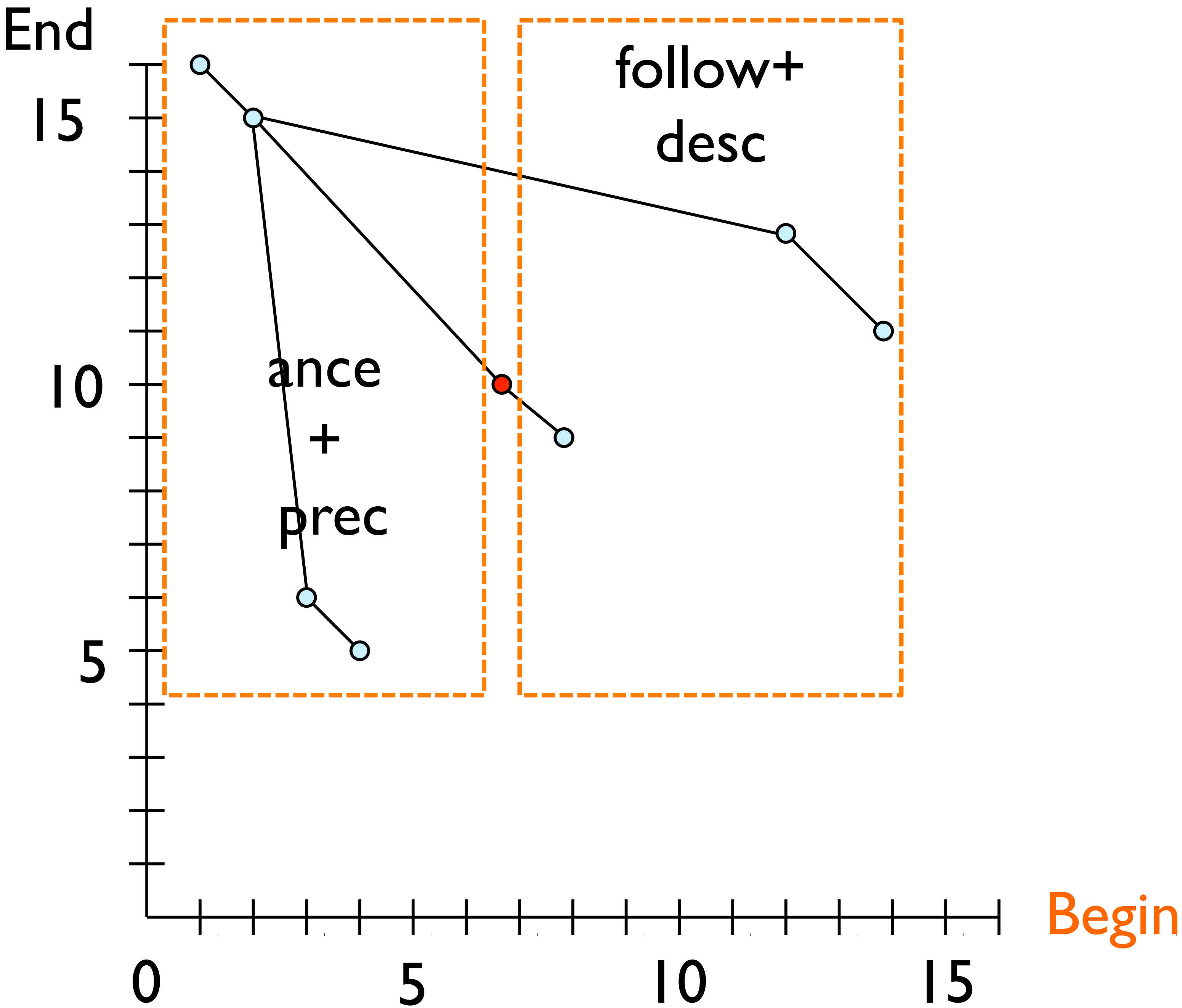
Begin/end numbering

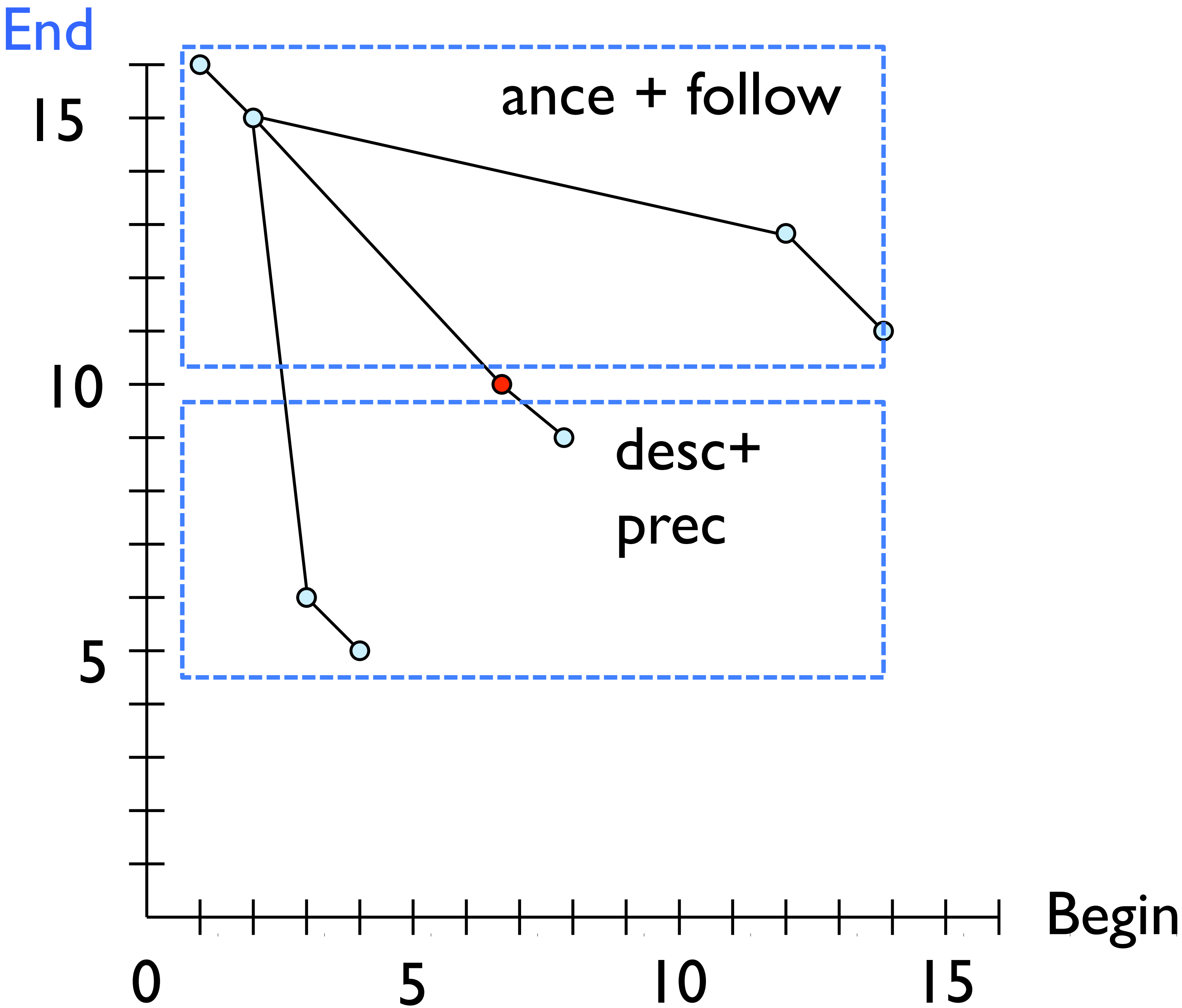


NODE Table

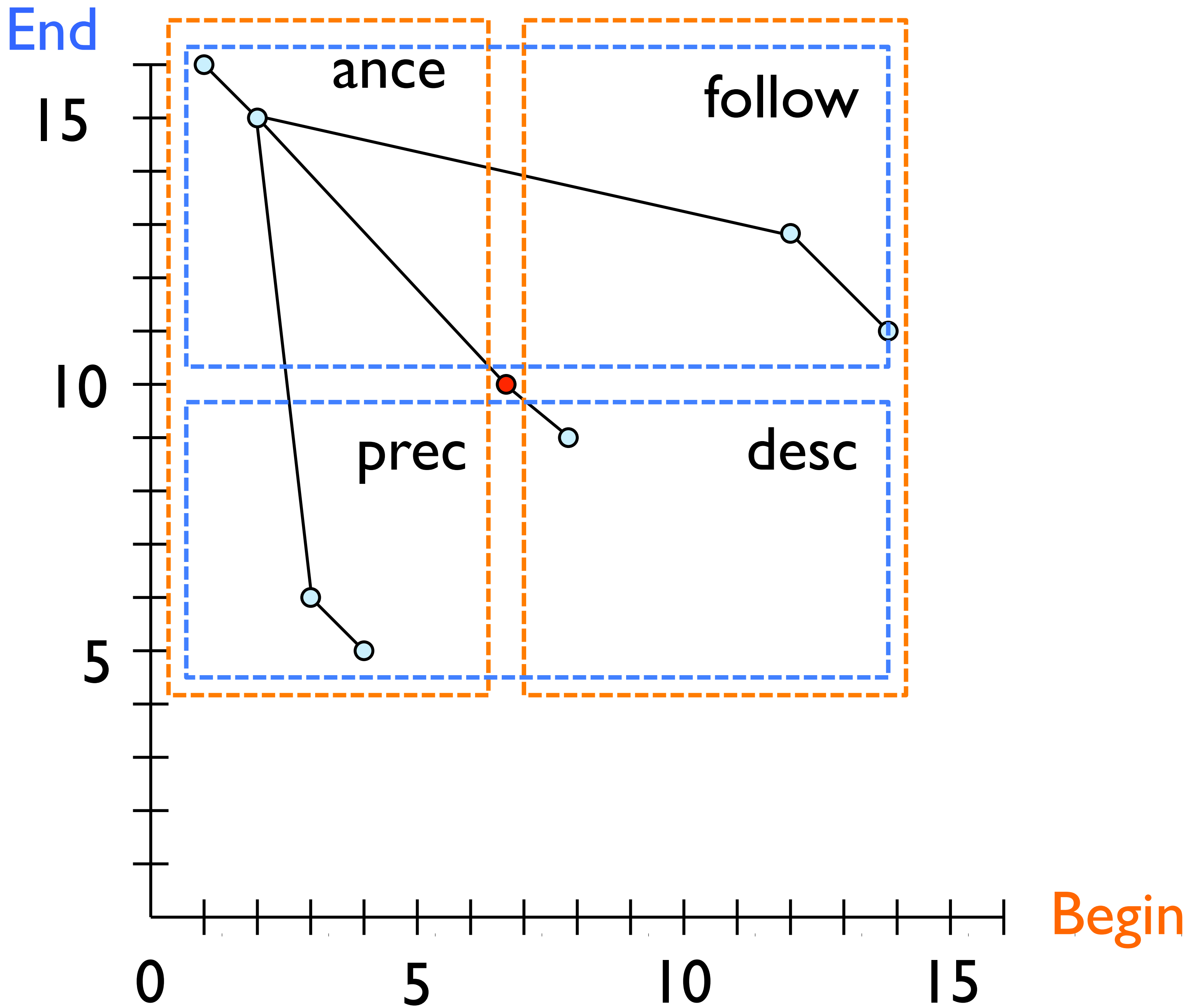
begin	end	par	tag	type
1	16		db	ELT
2	15	1	book	ELT
3	6	2	title	ELT
4	5	3		TEXT
7	10	2	author	ELT
8	9	7		TEXT
11	14	2	author	ELT
12	13	11		TEXT


End
15
10
5
0
5
10
15
Begin


End
15
10
5
follow+
desc
ance
+
prec
0
5
10
15
Begin



End
15
10
5
ance + follow
desc+
prec
0
5
10
15
Begin


End
15
10
5
0
5
10
15
Begin
ance
follow
prec
desc

From Axes to Intervas

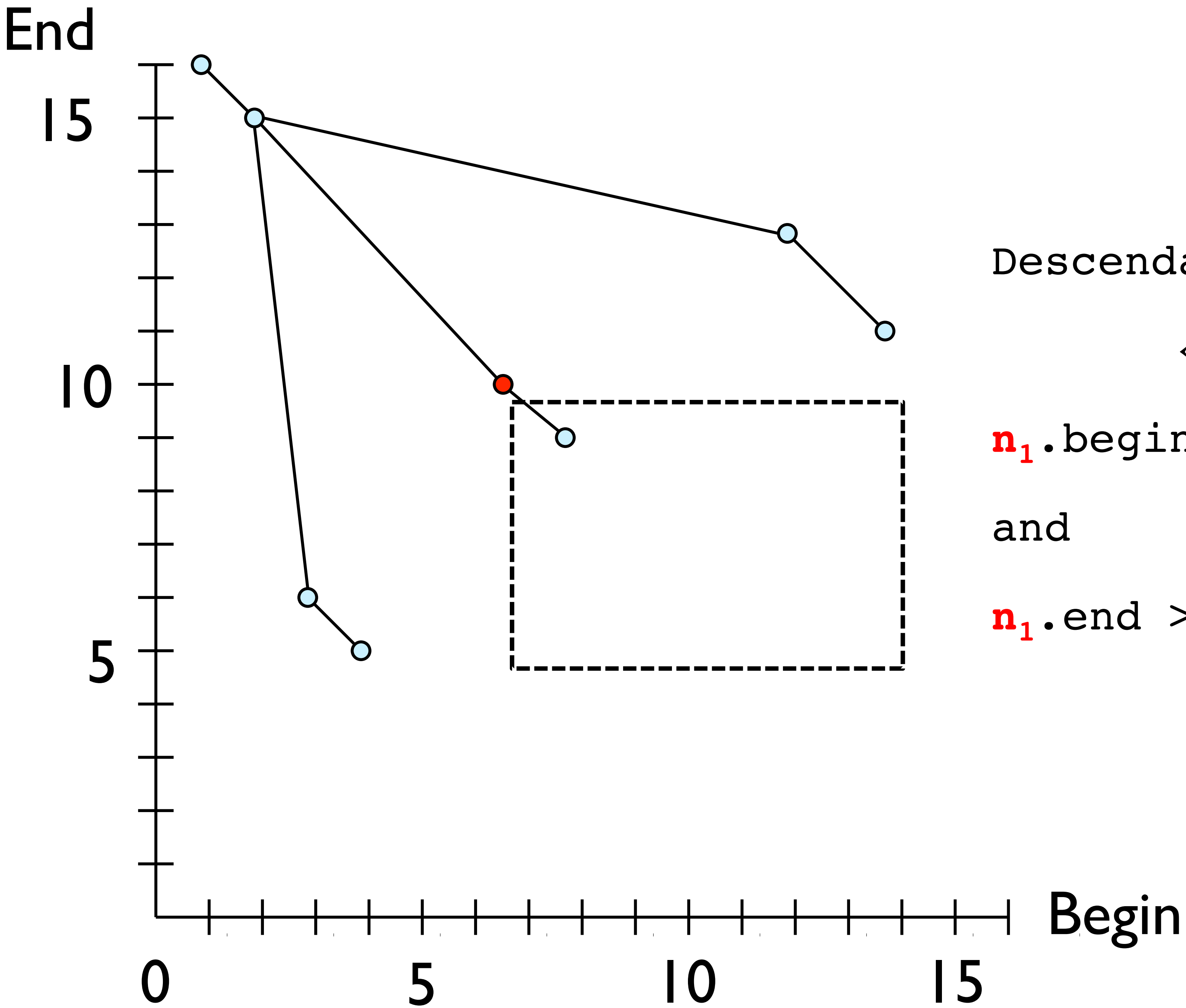
`Child(n1,n2)`

$\Longleftrightarrow$

`n1.begin = n2.par`

NODE Table

begin	end	par	tag	type
1	16		db	ELT
2	15	1	book	ELT
3	6	2	title	ELT
4	5	3		TEXT
7	10	2	author	ELT
8	9	7		TEXT
11	14	2	author	ELT
12	13	11		TEXT



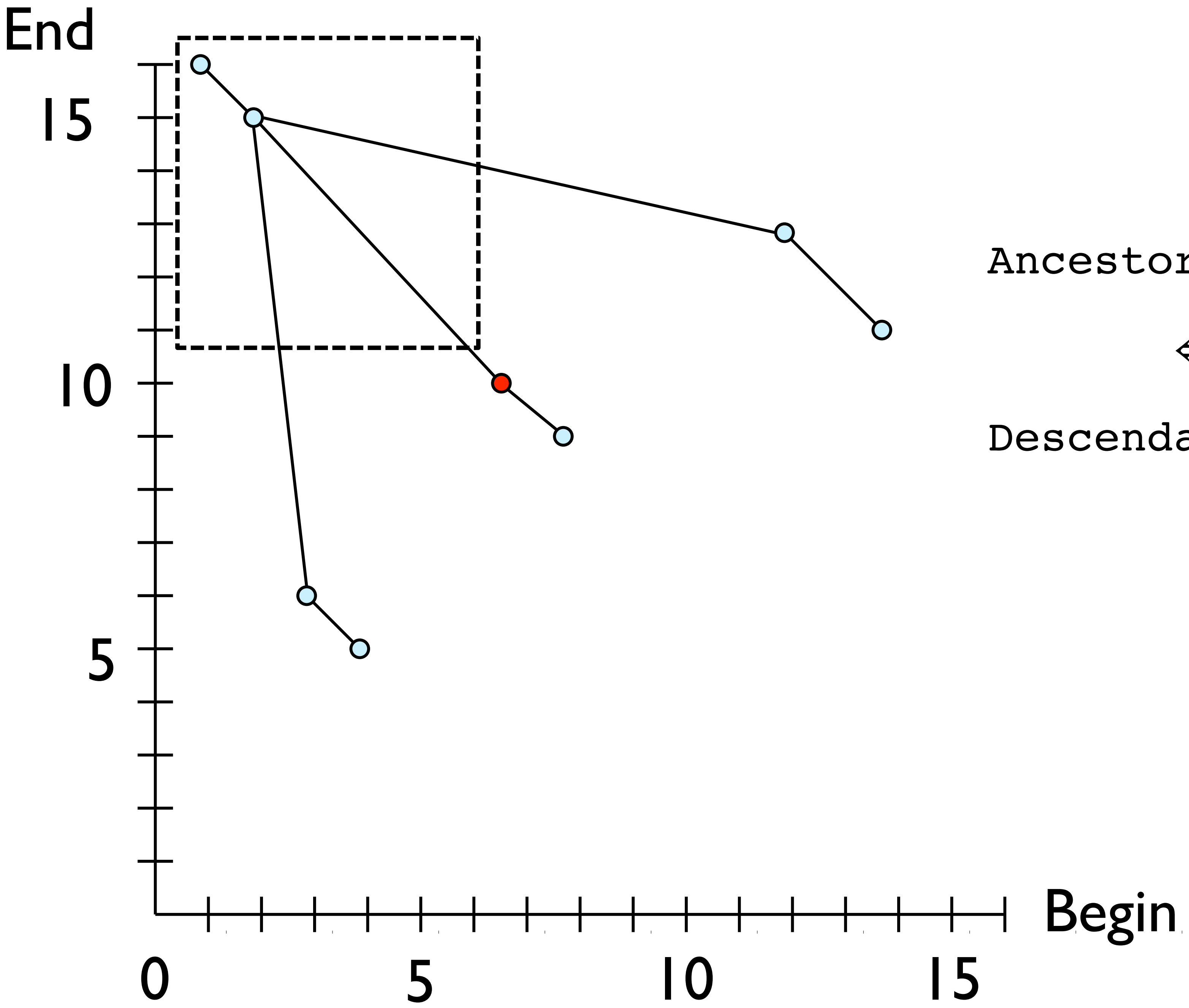
Descendant($\mathbf{n_1}$,$\mathbf{n_2}$)

$\Longleftrightarrow$

$\mathbf{n_1}$.begin < $\mathbf{n_2}$.begin

and

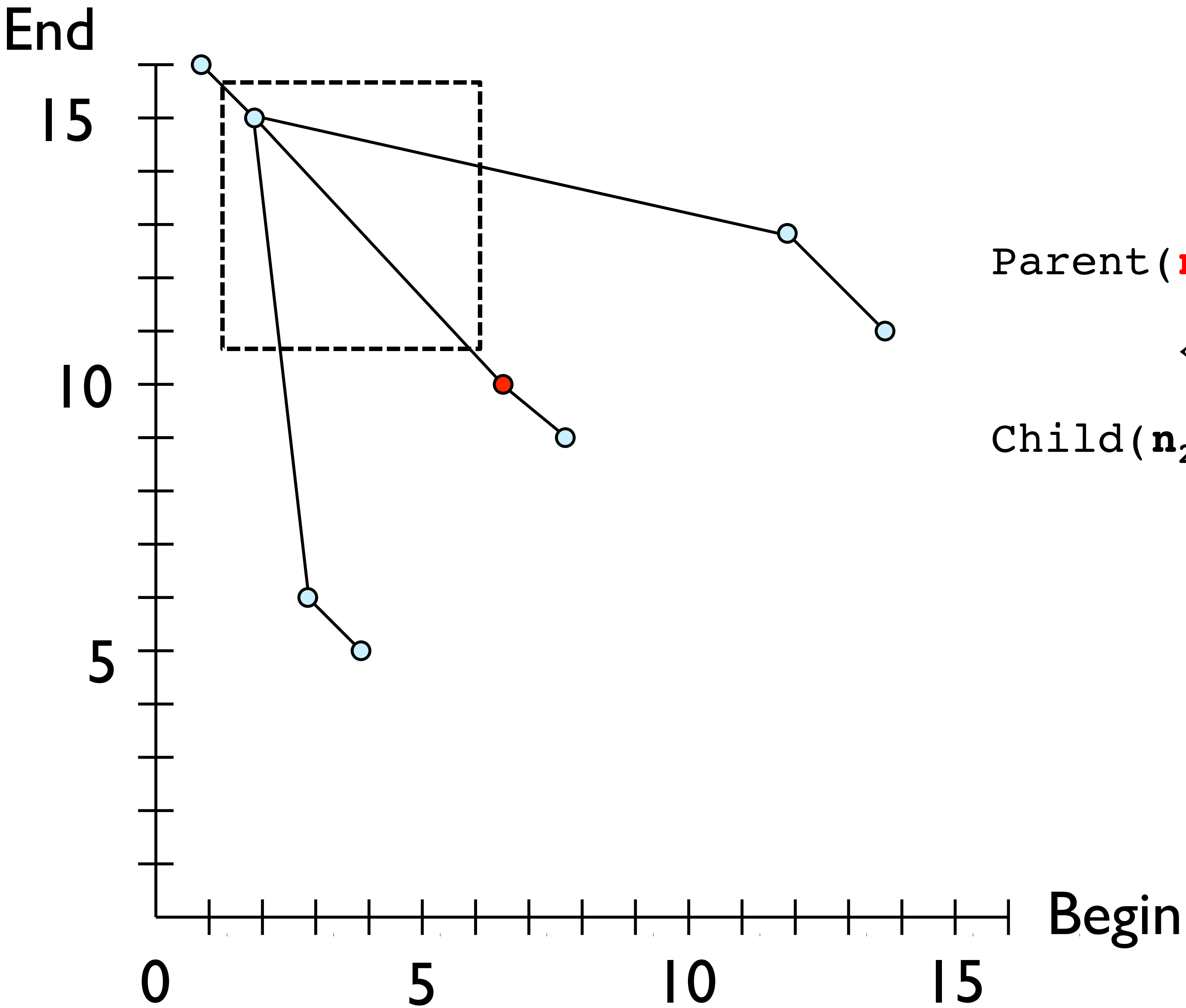
$\mathbf{n_1}$.end > $\mathbf{n_2}$.end



$\texttt{Ancestor}(\mathbf{n_1},\mathbf{n_2})$

$\Longleftrightarrow$

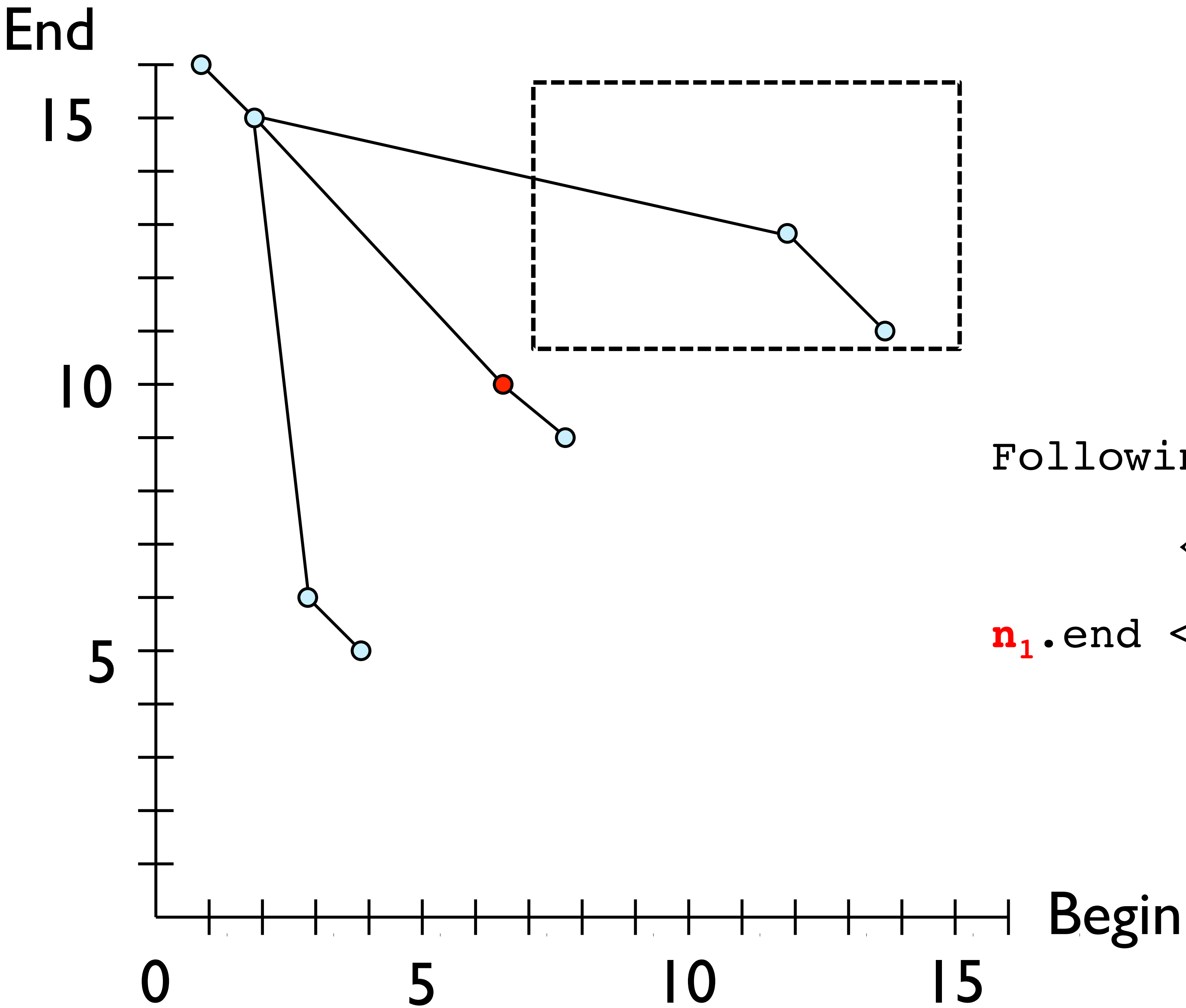
$\texttt{Descendant}(\mathbf{n_2},\mathbf{n_1})$



$\texttt{Parent}(\mathbf{n_1},\mathbf{n_2})$

$\Longleftrightarrow$

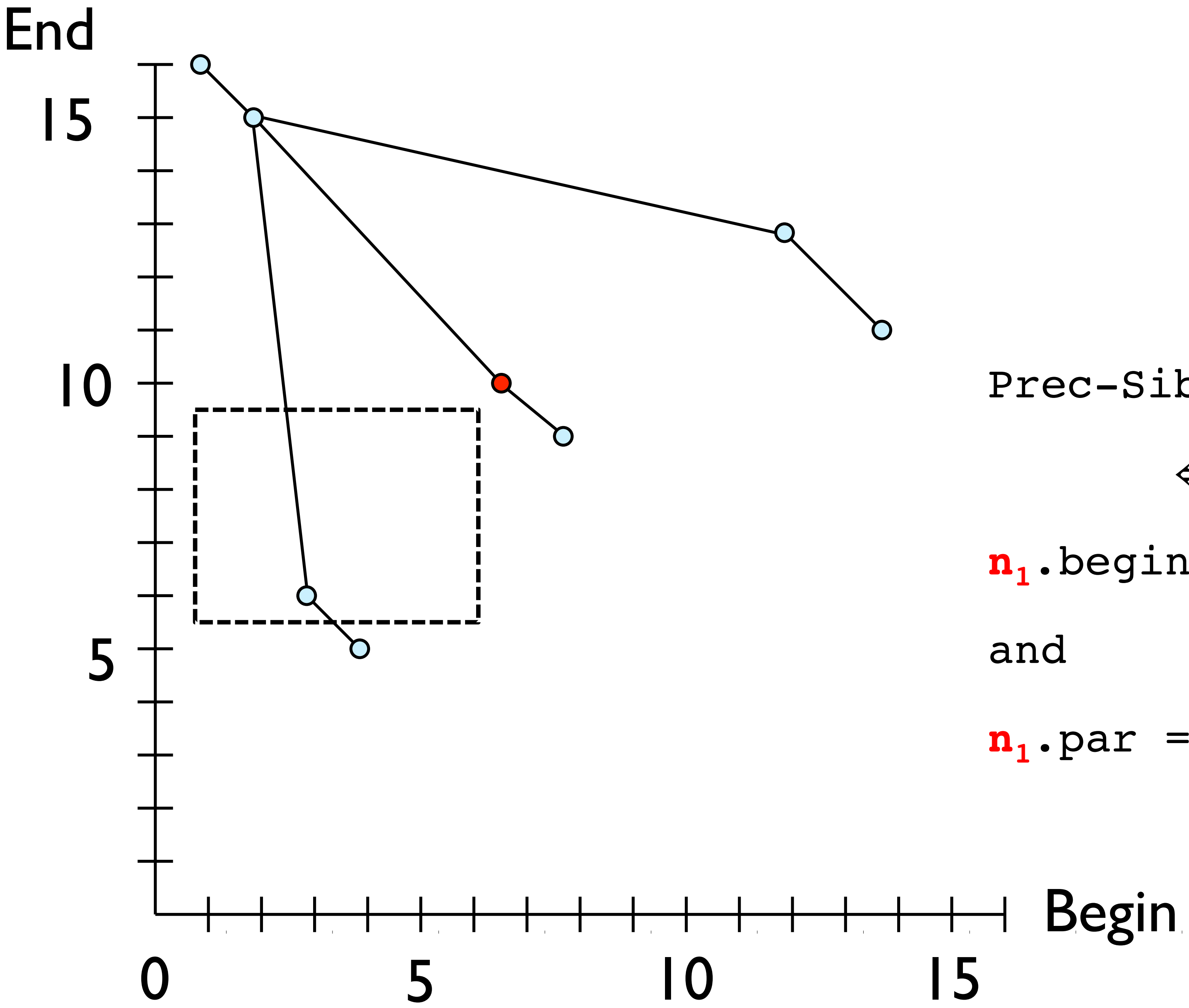
$\texttt{Child}(\mathbf{n_2},\mathbf{n_1})$



Following(n_1,n_2)

$\Longleftrightarrow$

n_1.end < n_2.begin



$\texttt{Prec-Sib}(\mathbf{n}_1, \mathbf{n}_2)$

$\Longleftrightarrow$

$\mathbf{n}_1.\texttt{begin} > \mathbf{n}_2.\texttt{end}$

and

$\mathbf{n}_1.\texttt{par} = \mathbf{n}_2.\texttt{par}$

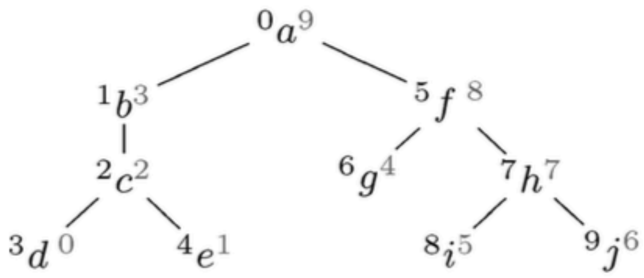
Ready to Query (with all axes!)

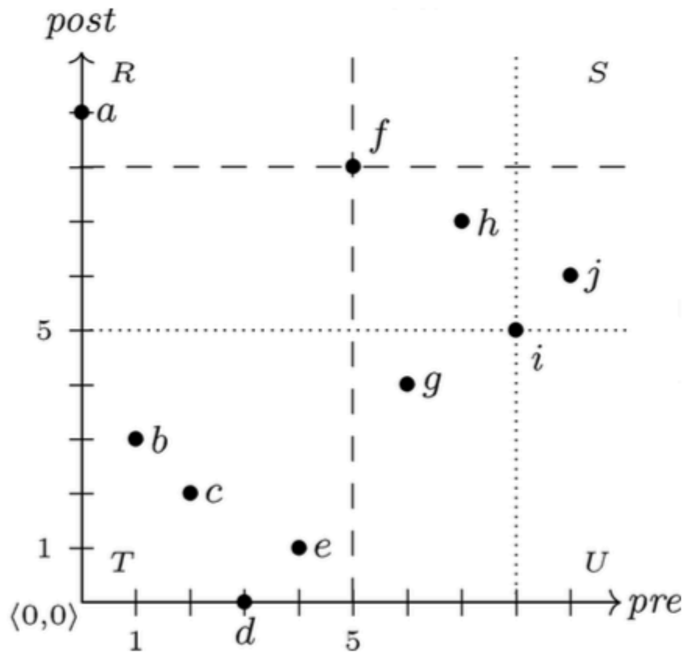
```
Q =  //a//b/ancestor::c//d/following-sibling::e

SELECT e.nodeID

FROM node a, node b, node c, node d, node e

WHERE
   a.tag = 'a', b.tag = 'b’,
   c.tag = ‘c’, d.tag=‘d’, e.tag=‘e’

  AND Descendant(a.nodeID,b.nodeID)
  AND Ancestor(b.nodeId,c.nodeId)
  AND Descendant(c.nodeId,d.nodeId)
  AND Following-Sibling(d.nodeId,e.nodeId)
```

to simplify the query, we assume that the nodes have also a unique **`nodeId`**

Other Approaches: Pre/Post

(Gurst et al. 2004)



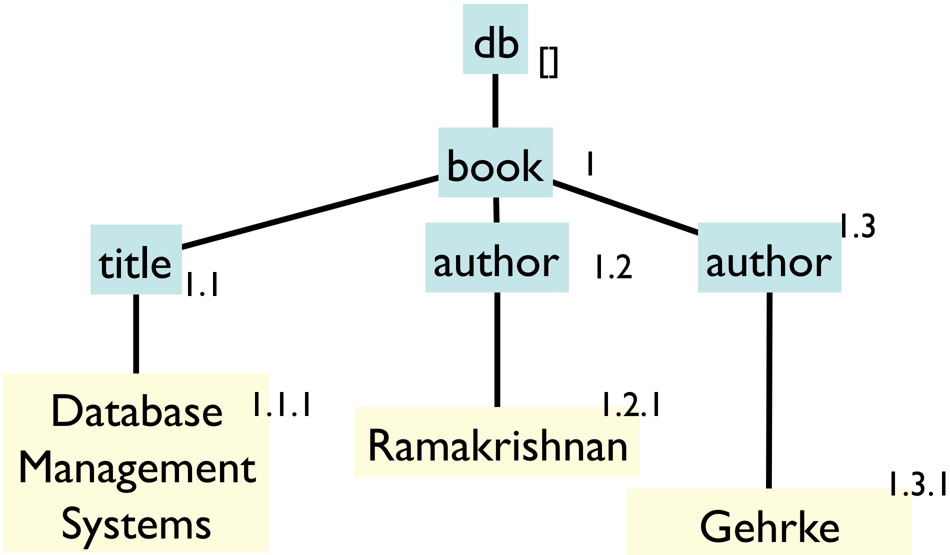


Replaces Begin/End with Pre/Post visit of the tree

Other Approaches: Dewey Decimal Encoding

Each node's ID is a list of integers

- $[i_1, i_2, \ldots, i_n]$ (often written $i_1.i_2.\ldots.i_n$)
- giving the "path" from root to this node



nodeID	tag	type
[]	db	ELT
1	book	ELT
1.1	title	ELT
1.1.1		TEXT
1.2	author	ELT
1.2.1		TEXT
1.3	author	ELT
1.3.1		TEXT

Summary

Dewey: string index, requires PREFIX, LEN UDFs

Interval: integer begin/end, pre/post indexes, only requires arithmetic

What about updates?

- Dewey: requires renumbering (exist update-friendly variants)
- Interval encoding: can require re-indexing most of the document

SCHEMA-AWARE XML STORAGE

Derivation of relational schema from DTD

Should be lossless

- the original document can be effectively reconstructed from its relational representation

Should support querying

- XML queries should be able to be rewritten to efficient relational queries

A book DTD

Complex
Regular Expressions

```
<!ELEMENT db  (book*)>

<!ELEMENT book (title,author*,chapter*, ref*)>

<!ELEMENT chapter  (text | section)*>

<!ELEMENT ref  book>

<!ELEMENT title  #PCDATA>

<!ELEMENT author  #PCDATA>

<!ELEMENT section  #PCDATA>

<!ELEMENT text  #PCDATA>
```

Recursion

Recall :regular expressions

`r`	`::=`	ϵ	empty sequence
	|	`a`	(tag element name)
	|	`(r,s)`	sequential composition
	|	`(r|s)`	union
	|	`(r*)`	repetition

`r+ = r*,r`   `r? = r|`ϵ

First-step : Simplification of RegExp

```
r ::=  ϵ
   |     a
   |   (r,s)          ← Order does not matter
   |   (r|s)
   |   (r1,r2)*       ← Correlation does not matter
```

```
(a,b)*  --1->  (a*,b*)  --2->  (a*|b*)
```

A book DTD

```
<!ELEMENT book (title,authors*,chapter*,ref*)>

<!ELEMENT chapter  (text | section)*>
```

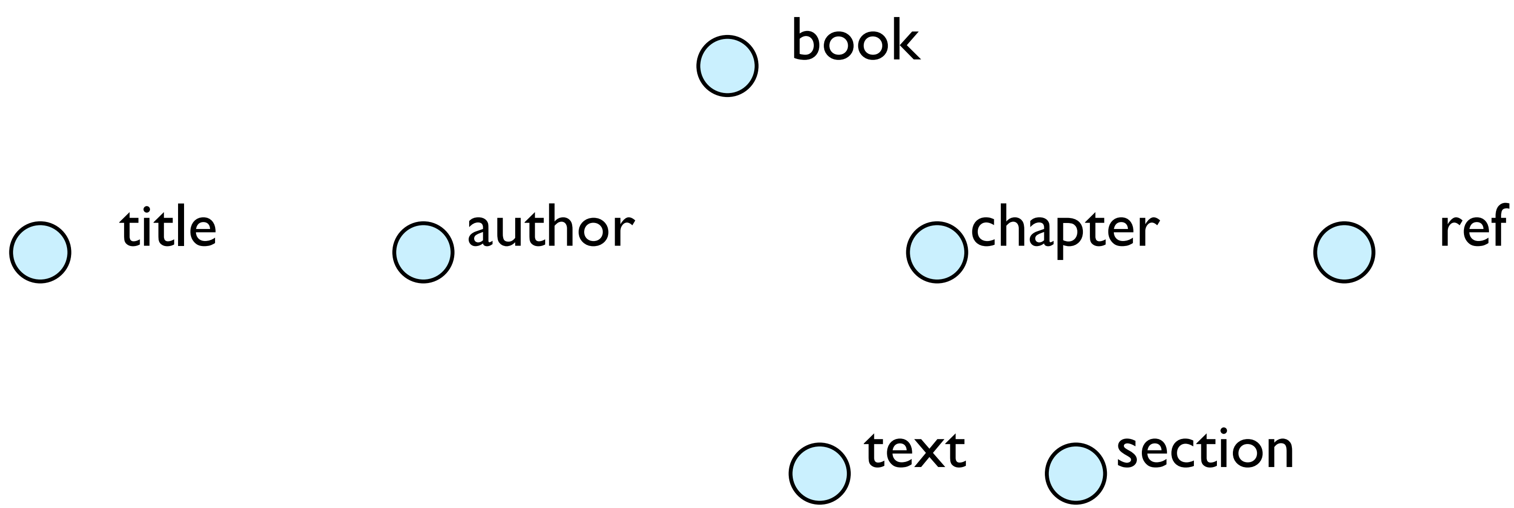
is approximated by

```
<!ELEMENT book (title|authors*|chapter*| ref*)>

<!ELEMENT chapter  (text*) | (section*) >
```

still correct, but less precise

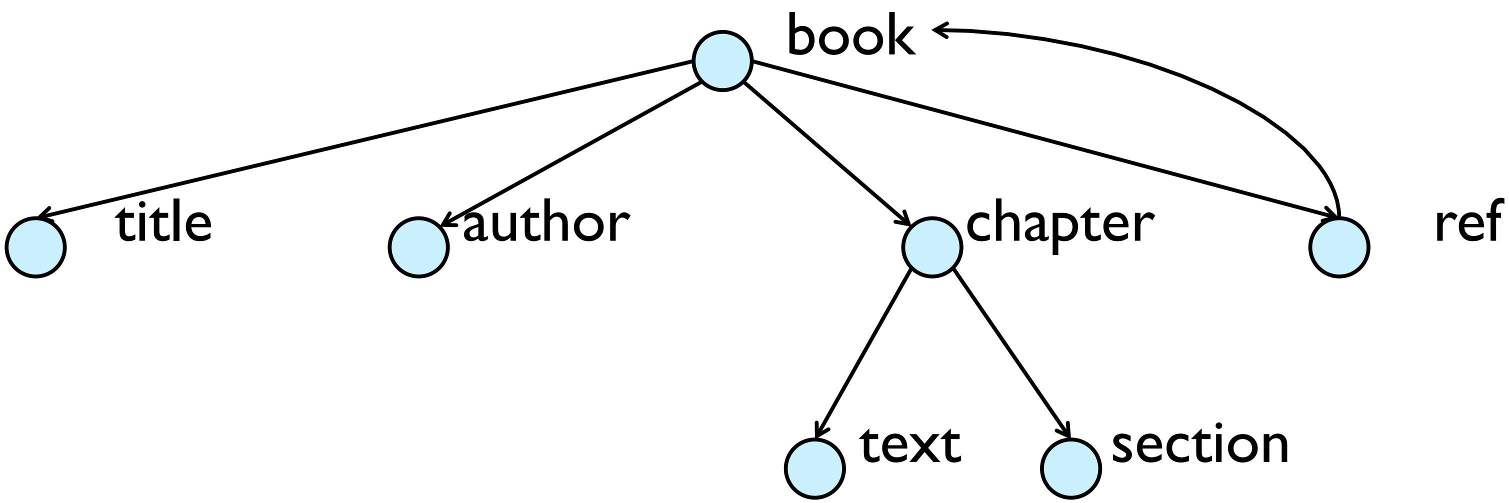
Second step: create a graph representation of the DTD

```
<!ELEMENT book (title|author*|chapter*| ref*)>
<!ELEMENT chapter  (text*) | (section*) >
<!ELEMENT ref  book>
```



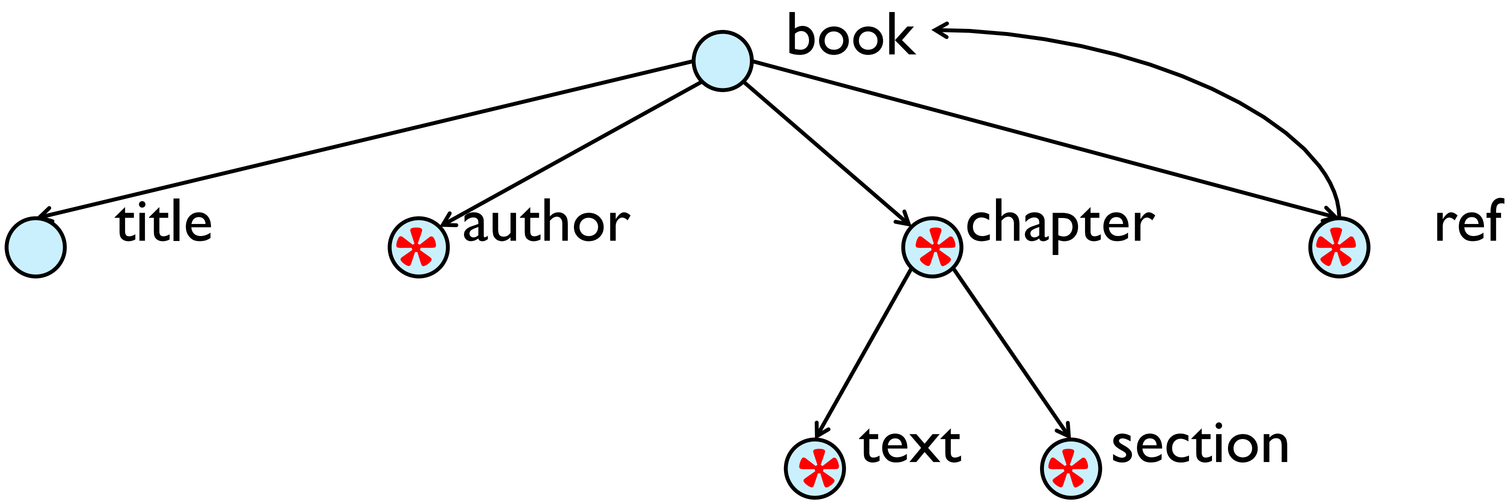
Second step: create a graph representation of the DTD

```
<!ELEMENT book (title|author*|chapter*| ref*)>

<!ELEMENT chapter  (text*) | (section*) >

<!ELEMENT ref  book>
```



Second step: create a graph representation of the DTD

```
<!ELEMENT book (title|author*|chapter*| ref*)>

<!ELEMENT chapter  (text*) | (section*) >

<!ELEMENT ref  book>
```



Graph representation of DTD

1. Each element type / attribute is represented by a unique node

2. Edges represent the subelement (and attribute) relations

3. Symbol *: denotes 0 or more occurrences of subelements

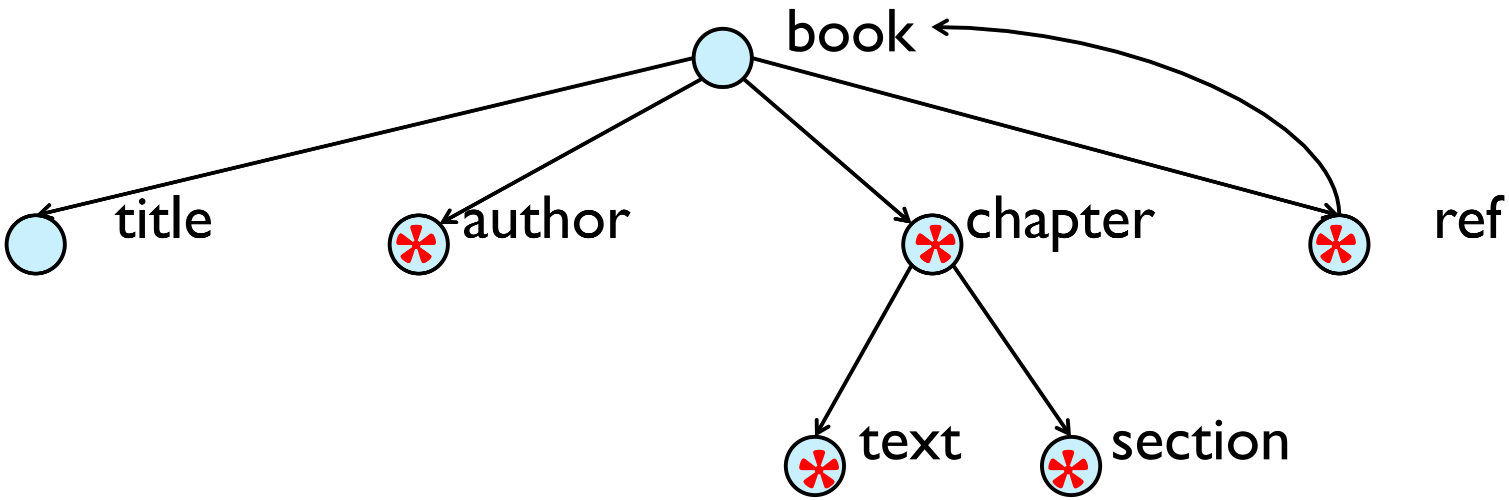
4. Cycles indicate recursion
    1. e.g., `book -> ref -> book -> ref`

Third step: Create Relations + Inline

Traverse the DTD graph depth-first and create relations for

(1) the root (2) each * node (3) each recursive node
(4) each node with at least 2 parents

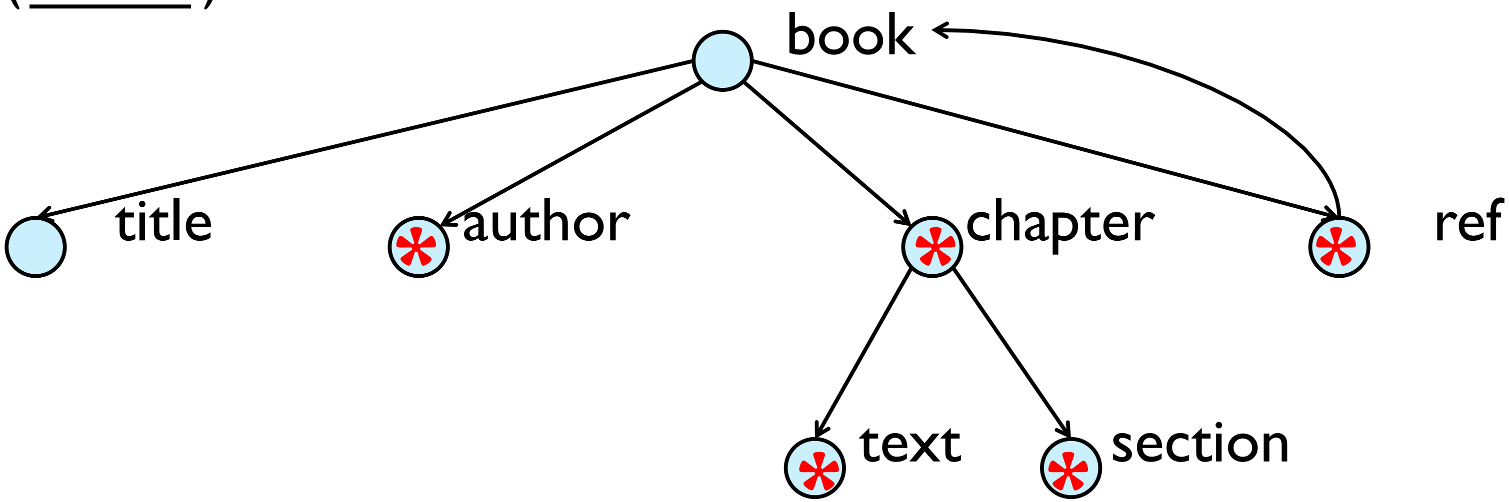
Nodes (w/out * and w/ only1parent) are inlined as fields: no relation created



Third step: Create Relations + Inline

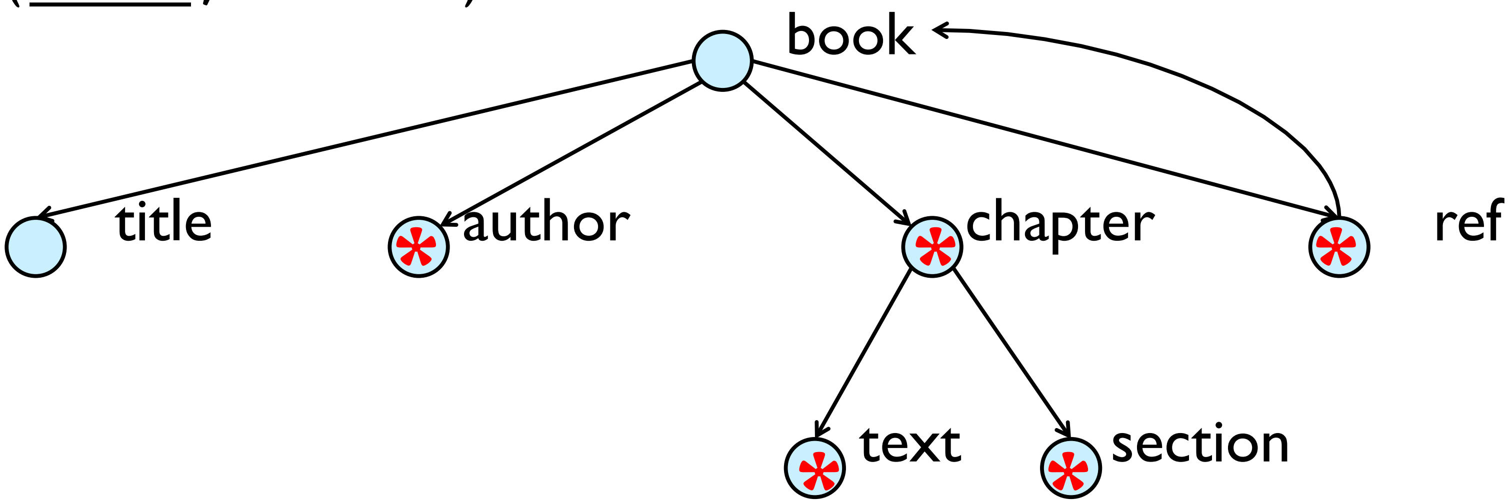
```
book(bookID, title: string)
author(authorID, author: string)
chapter(chapterID)
text(textID, text: string)
section(sectionID, section: string)
ref(refID)
```

we forgot something..



Third step: Create Relations + Inline

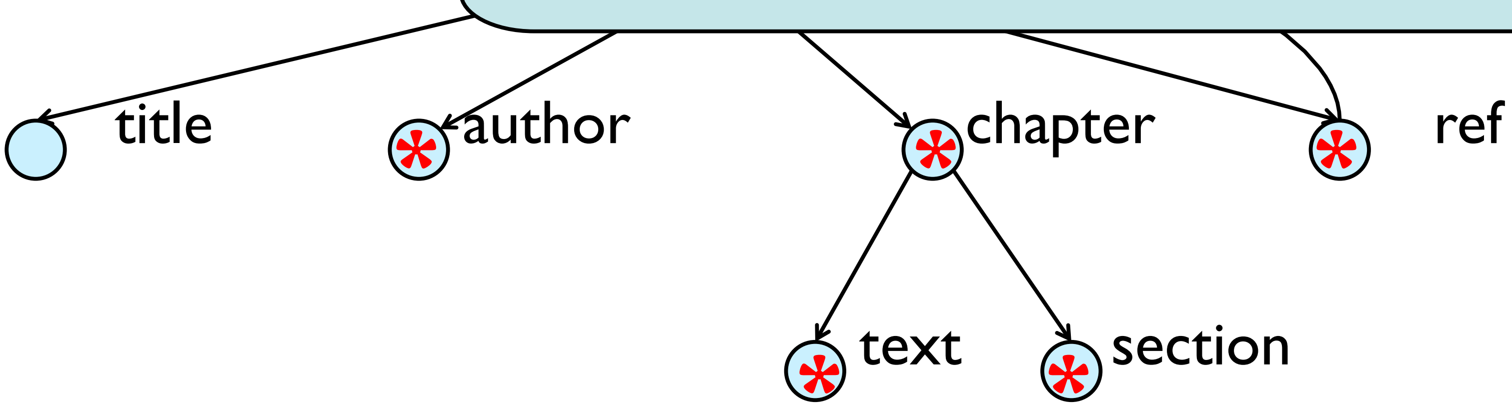
```
book(bookID, title: string)
author(authorID, bookID, author: string)
chapter(chapterID, bookID)
text(textID, chapterID, text: string)
section(sectionID, chapterID, section: string)
ref(refID, bookID)
```



Still missing detail : parent-ambiguity

```
book(bookID, code, title: string)
author(authorID, bookID, author: string)
chapter(chapterID, bookID)
text(textID, chapt
section(sectionID
ref(refID, bookID
```

code needed to distinguish book and `ref` parents



Still missing detail : parent-ambiguity

```
book(bookID, flag, title: string)
author(authorID, bookID, author: string)
chapter(chapterID, bookID)
text(textID, chapterID, text: string)
section(sectionID, chapterID, section: string)
ref(refID, bookID)
```

Foreign keys:

`book.parentID` ⊆ `db.dbID` if **`flag`**= 1

`book.parentID` ⊆ `ref.refID` if **`flag`**= 0

ref

text

section

Relational schema

```
book(bookID, code, title: string)

author(authorID, bookID, author: string)

chapter(chapterID, bookID)

text(textID, chapterID, text: string)

section(sectionID, chapterID, section: string)

ref(refID, bookID)
```

To preserve the semantics

- ID: each relation has an artificial ID (key)
- parentID: foreign key coding edge relation
- We can also add column naming path in the DTD graph

Note: `title` is inlined

Summary of schema-ware XML

Use DTD/XML Schema to decompose document

Reorganization of regular expressions

- (text , section)* $\rightarrow$ text* | section*
- document order and type-correlations are lost

Querying: Supports a large class of common XML queries

- Fast lookup & reconstruction of inlined elements
- Systematic translation unclear

COMMERCIAL SOLUTIONS

Well, XML is just text, right?

Most databases allow CLOB (Character Large Object) columns - unbounded length string.

So you just store the XML text in one of these

Surprisingly popular

- and can make sense for storing "document-like" parts of XML data (eg HTML snippets)
- But not a good idea if you want to query the XML

SQL / XML

Instead of blindly using CLOBs.. extend SQL with XML features

- "XML" column type
- XPath or XQuery queries (or updates) on XML columns

Also surprisingly popular (DB2, IBM, Oracle)

- Pro: At least DB knows it's XML
- Pro: Part of SQL 2003 (SQL/XML extensions)
- Con: Frankenstein's query language

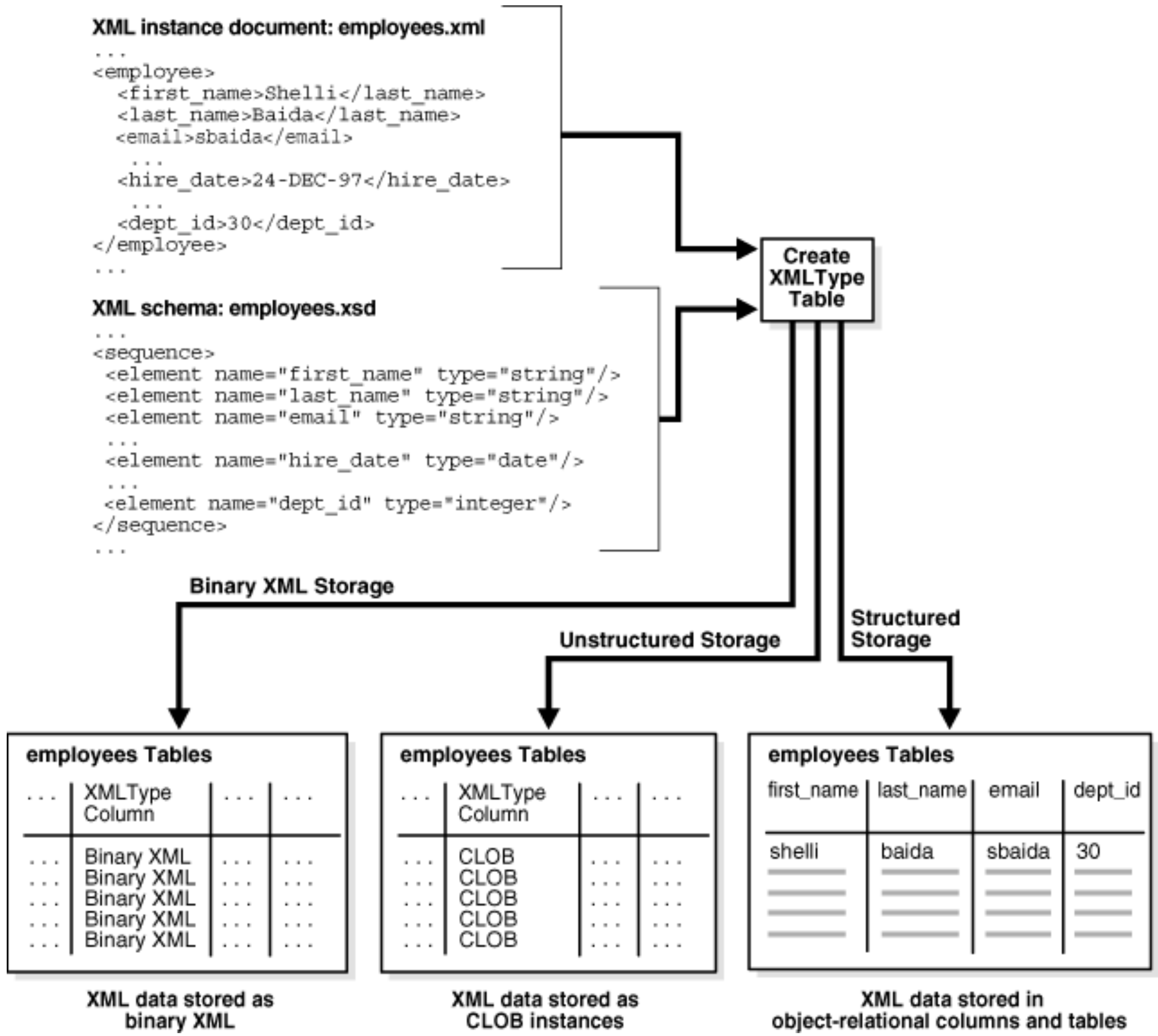
SQL/XML example

```
CREATE TABLE Customers(
  CustomerID int PRIMARY KEY,
  CustomerName nvarchar(100),
  PurchaseOrders XML, ...
)
```

SQL/XML example

```
SELECT CustomerName,
       query(PurchaseOrders,
   'for $p in /po:purchase-order
    where $p/@date < xs:date("2002-10-31")
    return <purchaseorder date="{$p/@date}">
             {$p/*}
           </purchaseorder>')
FROM  Customers
WHERE CustomerID = 42
```

XML Column Type : 3 Possible Storages



Oracle XML CLOB Storage

Simplest approach to implement and support

Byte fidelity : preserves original doc (even white space)

Performances

➚ load insert full retrieval

➘ query schema evolution

Need to parse the document for all XML processing (memory overhead)

Oracle XML Relational Storage (OR)

Schema-aware mapping to relational tables

Byte fidelity not always guaranteed

Performances

- ➚ query for highly-structured data
- ➘ query for un-structed data (many joins), full-retrieval
- ➘ flexibility schema evolution, load / insert / delete

Motivation/Goals for Binary XML

XML Schema usage

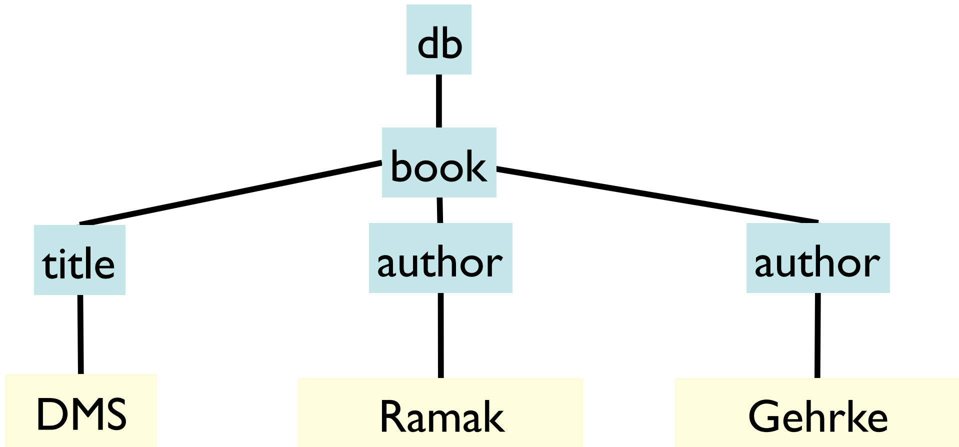
- Need to be efficient for query processing on schemaless & loosely structured schemas; use schema for optimization

Provide good performance for a wide range of operations

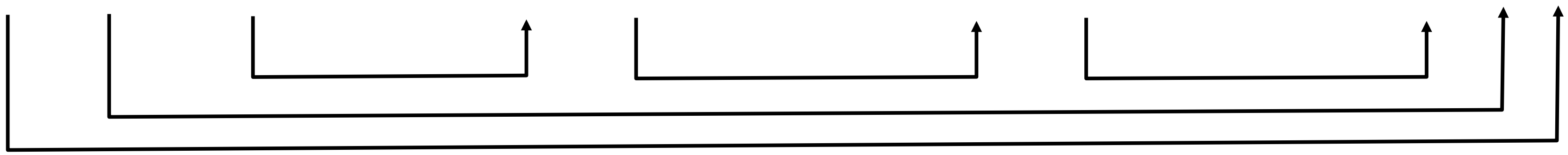
- Query
- DML: Insert/Load, Partial (piecewise) update
- Full-document & fragment retrieval
- Schema Validation & Evolution

What is binary XML ?

XML tree = balanced parenthesized expression



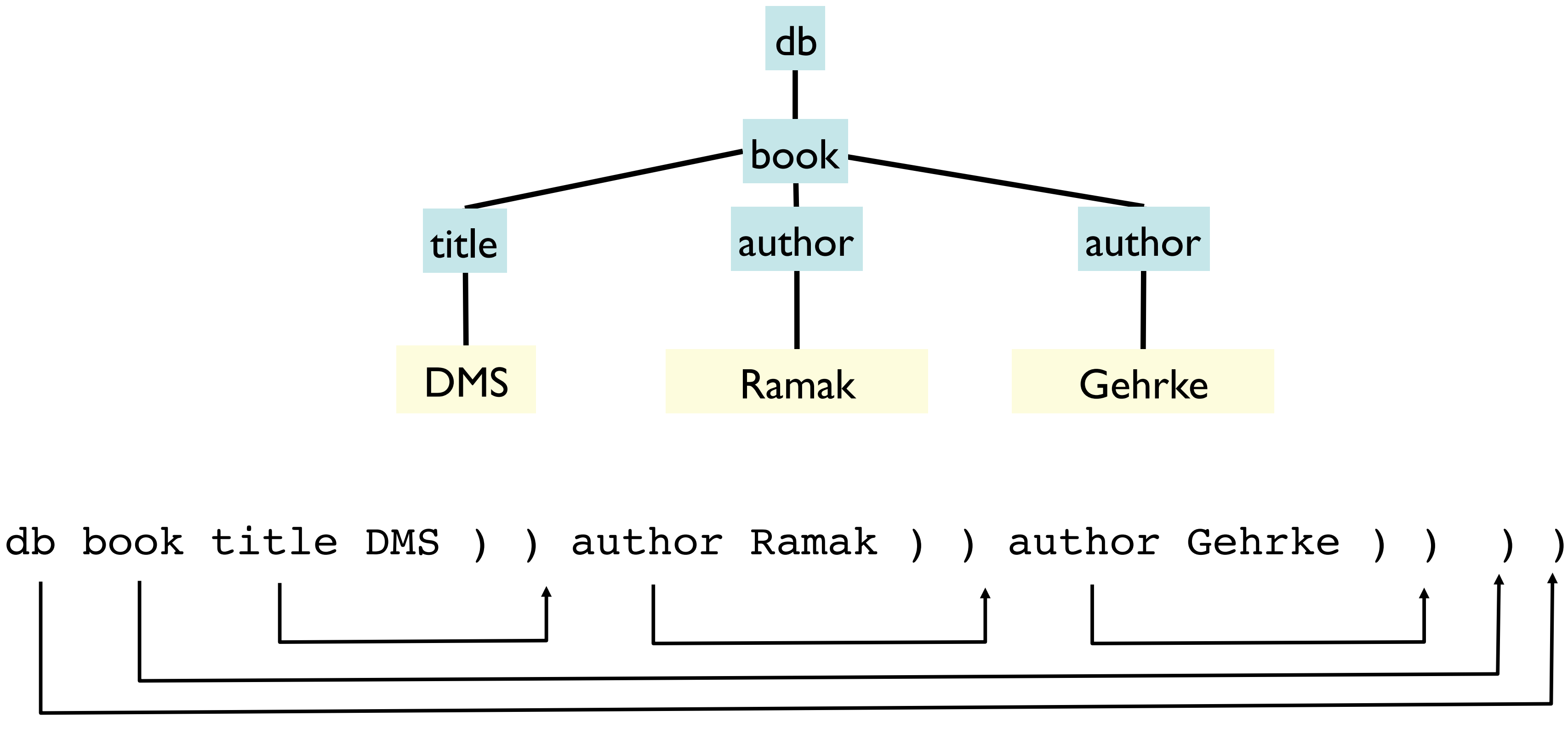

```
db book title DMS ) ) author Ramak ) ) author Gehrke ) )  ) )
```



What is binary XML ?

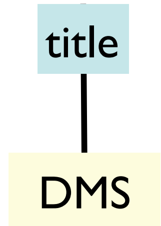
Navigation = find corresponding (open-)closed parenthesis

Update = insert / delete a substring (easy for Oracle's file system)



What is binary XML ?

BinaryXML is made of sections (subtrees)



`title DMS ) )`

Each section has an header recording (among others)

- path leading to that subtree
- sibling-position

Header Path=db.book.title Position = 1	OPEN- ELEMENT	**Header** Path=db.book.title.text() Position = 1	OPEN- TEXT	DMS	CLOSE- TEXT	CLOSE- ELEMENT

What is binary XML ?

- Query evaluation : can be done by scanning the headers

- Storage:
  - Compression of elements of the same type (e.g. many authors)
  - Further compression when schema is available
  - Also datatypes (int, string) can have best low-level storage

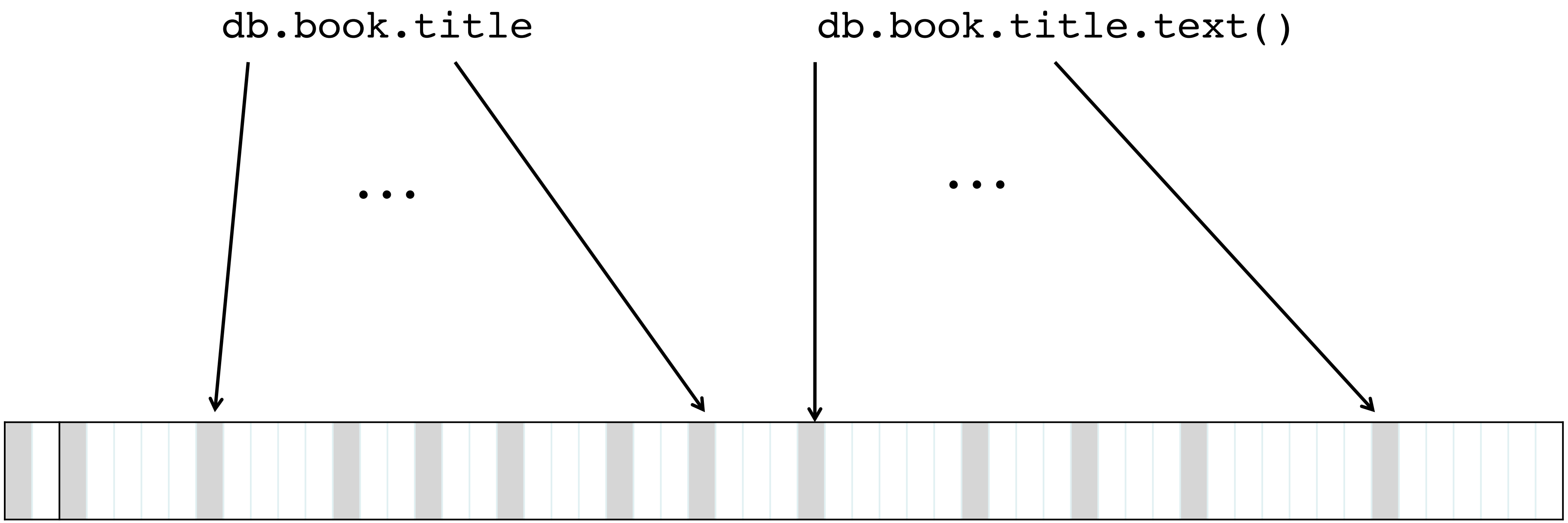
- Of course, every information (like paths) is encoded with an ID !

Header Path=db.book.title Position = 1	OPEN- ELEMENT	**Header** Path=db.book.title.text() Position = 1	OPEN- TEXT	DMS	CLOSE- TEXT	CLOSE- ELEMENT

What is binary XML ?

Add index to directly access the sections (in constant time).

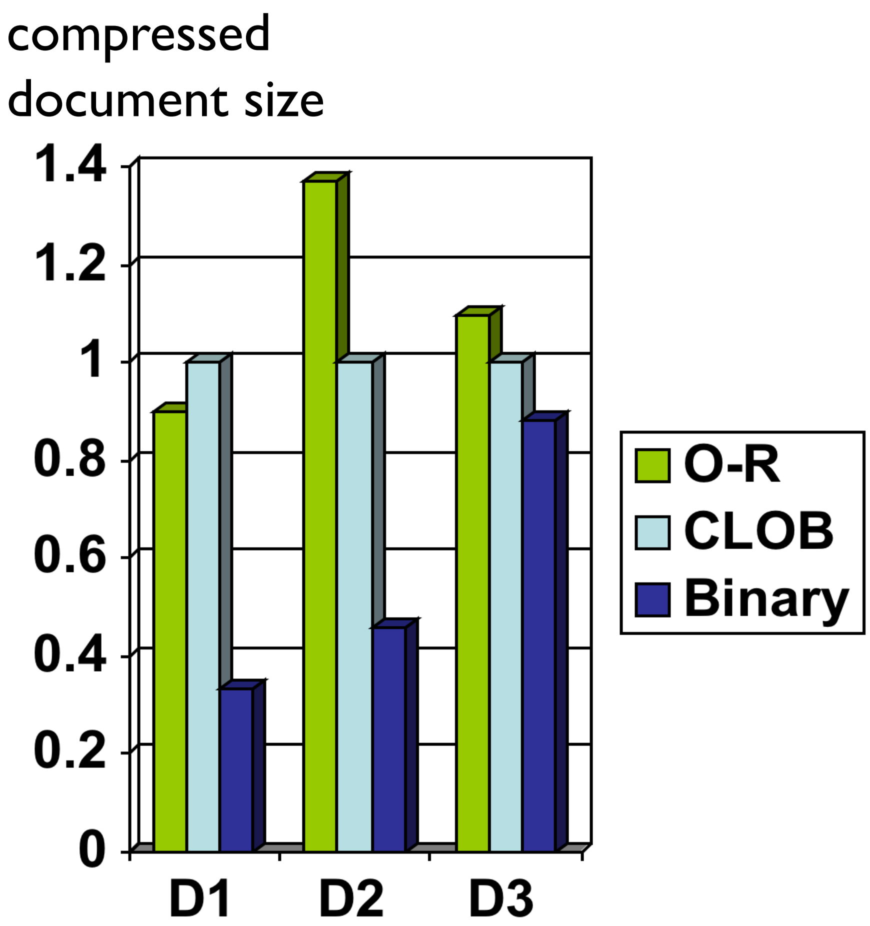
Dramatically improves performances (at the price of space).



Comparisons of storage models

	CLOB	OR	Binary XML
Query	poor	excellent	good/excellent
DML	poor	good/excellent	excellent
document retrieval	excellent	good/excellent	excellent
schema flexibility	good	poor	excellent
document fidelity	excellent	poor	good/excellent
mid-tier integration	poor	poor	excellent

Performance: Compression



D1 – Structured
D2 – Semi-structured
D3 – Document-centric

Based on actual ORACLE customer datasets

Mix of XML document sizes

ORACLE Customer use cases

	Data-Centric	Document-Centric	
Use Case	XML schema-based data, with little variation and little structural change over time	Variable, free-form data, with some fixed embedded structures	Variable, free-form data
Typical Data	Employee record	Technical article, with author, date, and title fields	Web document or book chapter
Storage Model	Object-Relational (Structured)	Binary XML	
Indexing	B-tree index	- XMLIndex index with structured and unstructured components - XML Full-Text index	- XMLIndex index with unstructured component - XML Full-Text index

FIGURE 1: XML USE CASES AND XMLTYPE STORAGE MODELS